

Recent Advances in Virtual Worlds for Science and Technology Research and Development

Walt Scacchi

Center for Computer Games and Virtual Worlds

Donald Bren School of Information and Computer Sciences

University of California, Irvine

<http://cgvw.ics.uci.edu>

Overview

- Recent virtual world projects for Science or Technology R&D
- Future opportunities for virtual worlds for science and technology R&D

Strategies for Creating Value with Virtual Worlds

- Creating game-based learning environments with virtual worlds
 - “Play” and experiential behavior are surprisingly effective way to audition, rehearse, act, fail, and learn
 - Mixed reality worlds can link virtual and physical activities
 - Virtual worlds are best at providing new *experiences*
 - Virtual work practices
 - Not the same as existing work practices
 - Need to learn what to do, how to do it, and more
 - Not obvious how to be faster, better, and cheaper using virtual worlds!

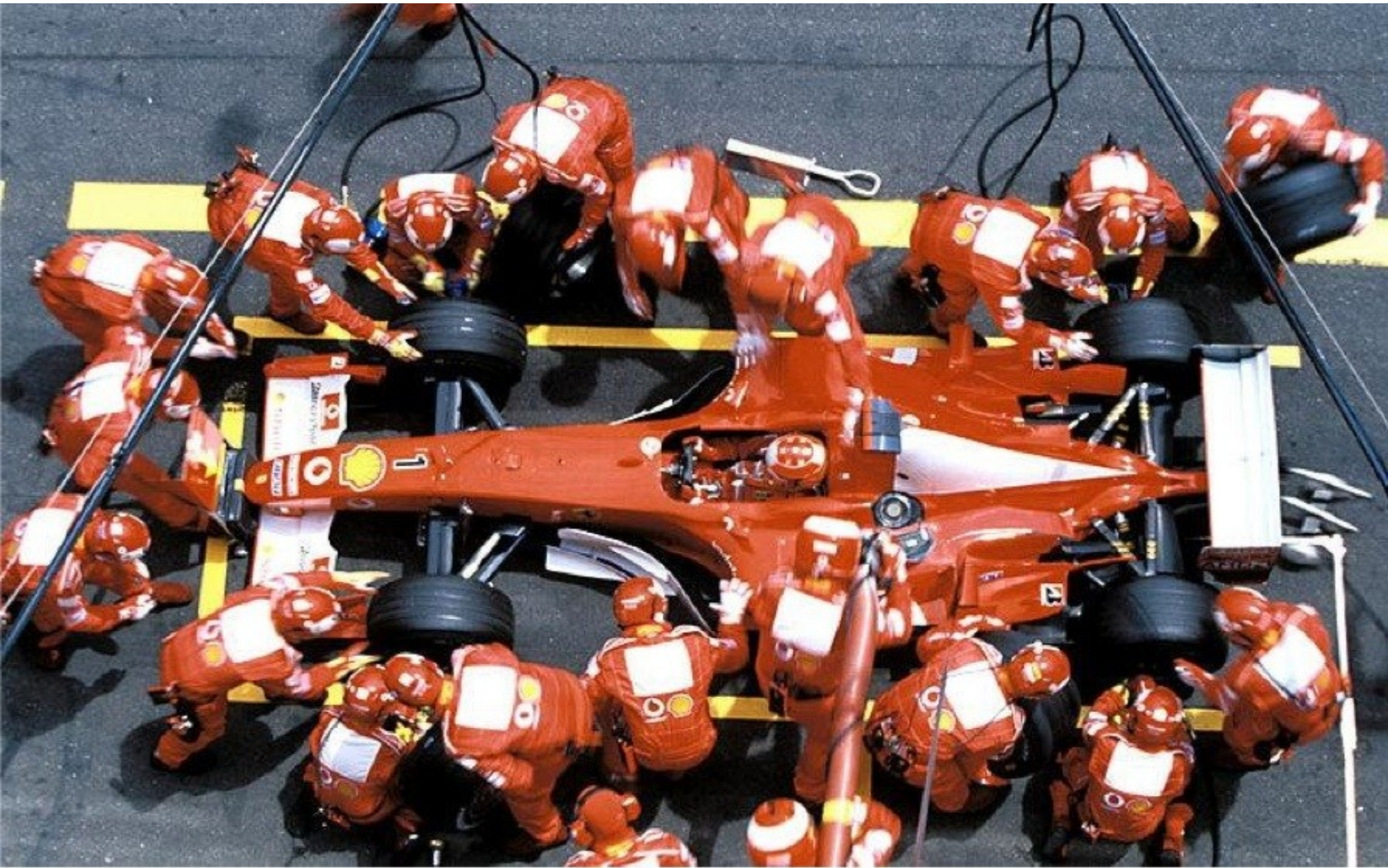
Collaborative meeting work in virtual world



Collaborative work in physical world



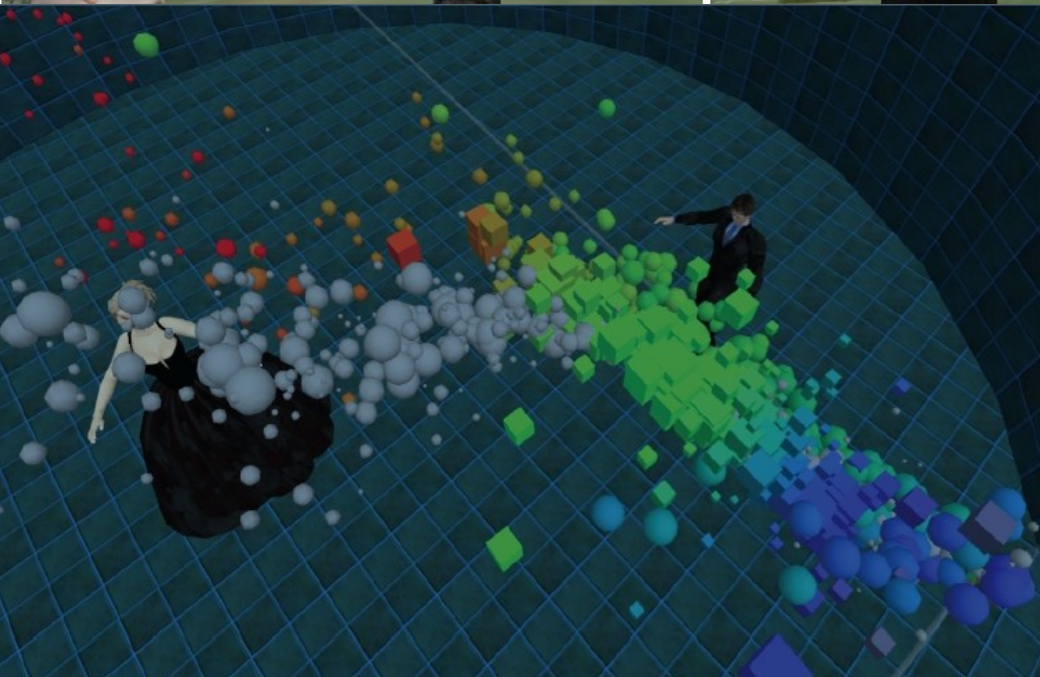
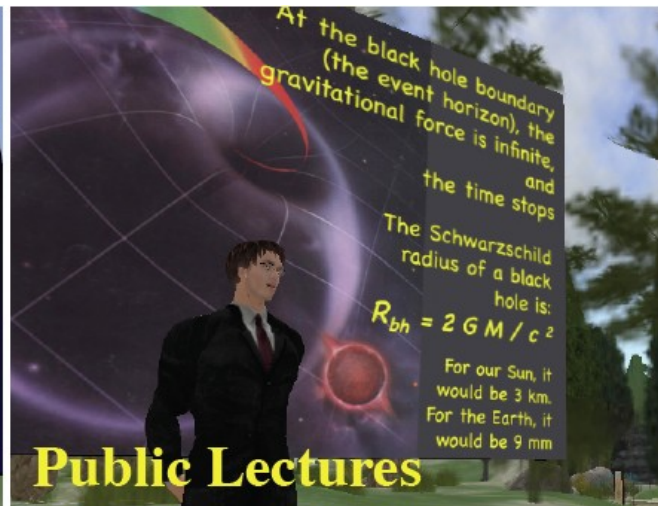
Radically colocated work in physical world



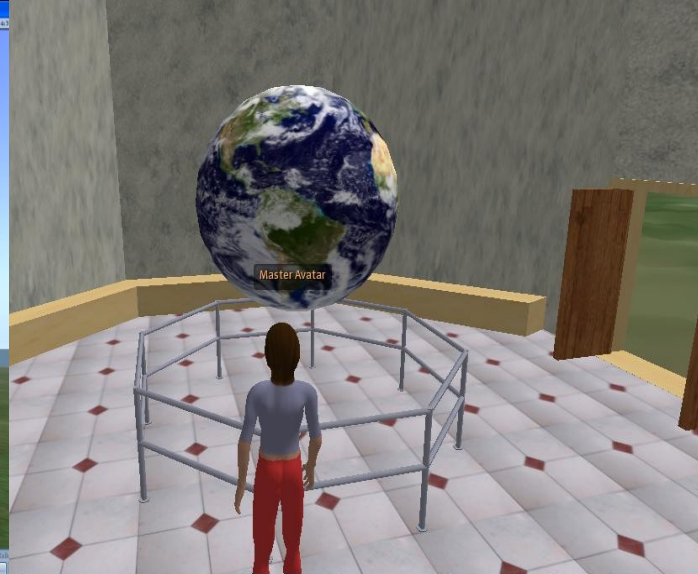
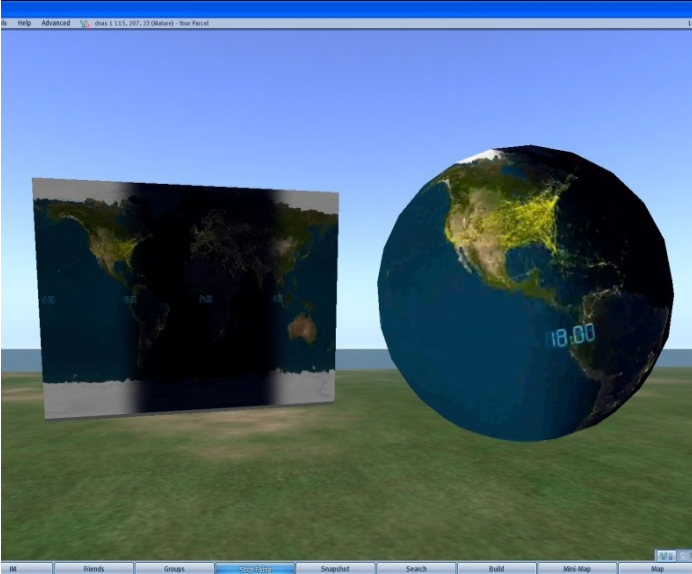
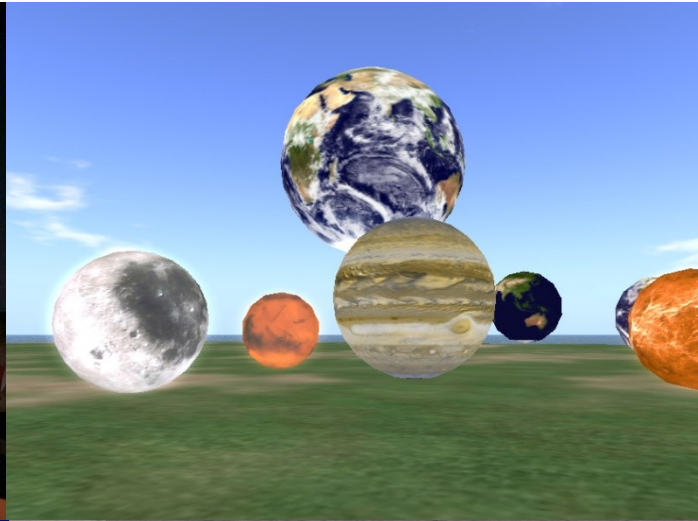
Recent Virtual World Projects for Science and Technology R&D

- Collaborative science meetings and immersive simulations
 - Meta Institute for Computational Astrophysics
- Collaborative science learning and data exploration environment with spherical displays at *Discovery Science Center* and in *OpenSim*
 - Science on a Sphere
- Collaborative game world for semiconductor fabrication or nanotechnology design
 - FabLab training simulator
- Game-based virtual worlds for advanced health care
 - Robotic therapeutics and tele-rehabilitation
- Envisioning future virtual worlds for possible cultural experiences and new technological innovation opportunities
 - Virtual Life 2010+
 - Immersive motorsports racing experiences
 - Low-cost to high-cost virtual world simulators
 - OutRun @ UCI

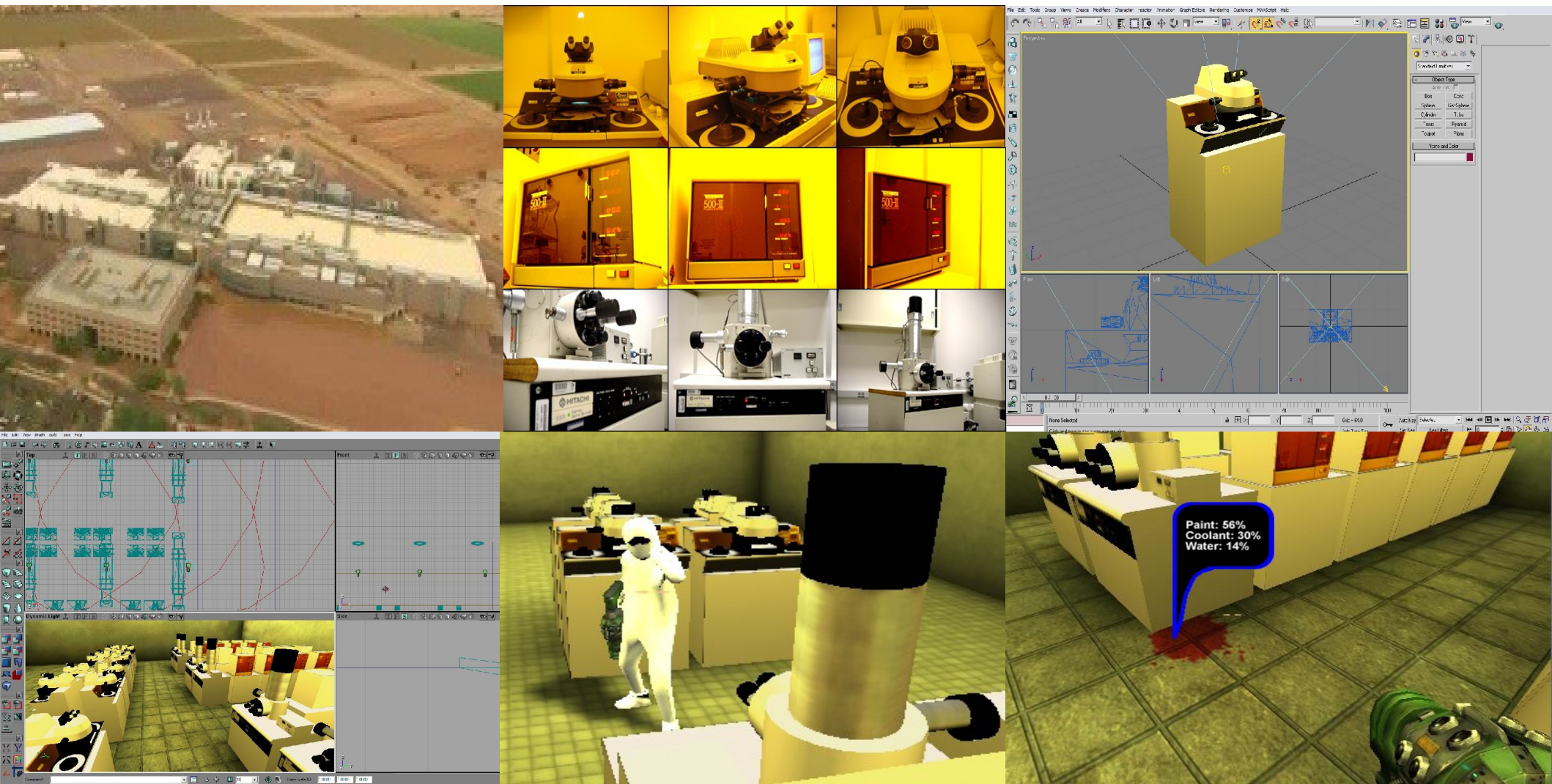
Virtual Worlds for Scientific Collaboration: *Meta Institute for Computational Astrophysics*



Spherical displays and “spherecasting” support: *NOAA Science on a Sphere* installation in *Opensim*



Game-based virtual world for semiconductor/nanotech fabrication training, remote presence and diagnosis



FabLab Demo Reel

Semiconductor/nanotechnology fabrication training game

working in a cleanroom

Suit made of
ultra clean material

Battery pack for
air filter system

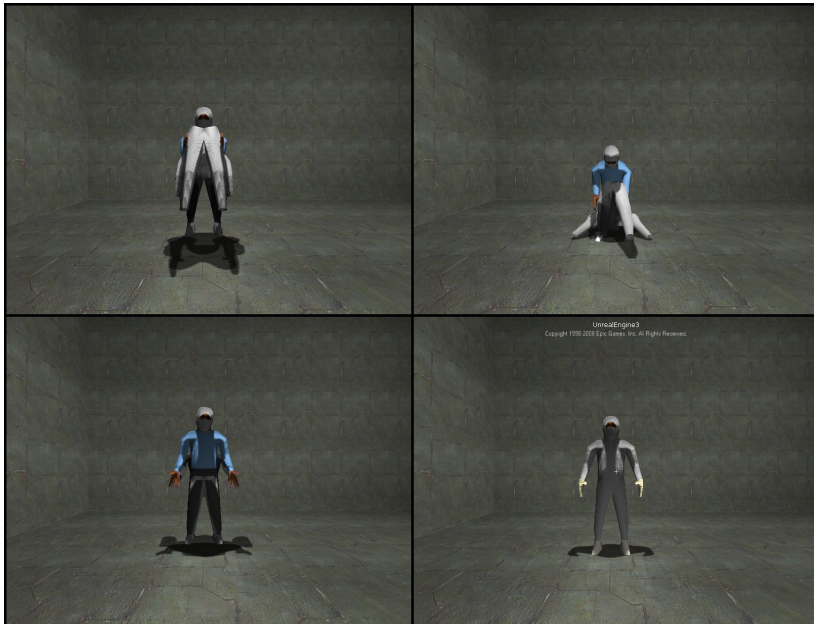
2 pairs of gloves
nylon & latex

2 pieces
of foot gear
disposable
shoe covers &
outer booties

Helmet
includes
air filter
unit

Will also
wear
hairnet
& safety
glasses

Belt



Virtual worlds for health care and tele-rehabilitation

- **Virtual worlds can be used to support various kinds of tele-medicine and tele-robotics applications/tasks**



- **“Rehabilitation” tasks supported can include:**
 - Remote observation, tele-consultation, role-playing and identity switching through avatars, device data collection, device software updates, collaborative product/prototype development, and more

Assisted performance training and robotic rehabilitation

- **Wii Sports** (best selling game for Nintendo in 2007; 45M copies sold worldwide through 2009)

- **Boxing**
- **Bowling**
- **Golf**
- **Tennis**
- **Baseball**



What's next?



VW haptic interfaces with therapeutic applications

- Simulated devices
 - Guitar Hero guitar; Rock Band drum set
- Haptic wheels, trackballs, and joysticks
- Force-feedback play controllers (racing game wheels, pneumatic bladders)
- Multi-sensor play controllers (including video capture, infra-red, accelerometers, neurological sensors, electro-goniometers (SEMG), etc.)
 - Wii Remote and nunchuk
- Multi-jointed, body-worn sensors as play controllers
 - Data gloves



- GypsyMIDI



Haptic interfaces with possible therapeutic applications

- Endoscopic surgery training “joysticks”

- Simball 4D joystick adapted to therapeutic game play for stroke rehabilitation

- <http://www.g-coder.com/content/view/7/6/>



- 3D, real-time video motion capture enabling *mixed reality game play* spanning physical and virtual worlds

- *Project Natal* at Microsoft
- In-game characters can interact with human players through gestures and body movements
- http://www.youtube.com/watch?v=g_txF7iETX0



Some findings on Games for Health/Therapeutic Applications

- The design and utility of a game to realize therapeutic value is not obvious.

E. Flores, G. Tobon, et al.,
Improving Patient Motivation in
Game Development for Motor
Deficit Rehabilitation, *ACM 2008
Intern. Conf. Advances in Computer
Entertainment*, 381-384.

Table 1. Gaming design criteria for stroke rehabilitation programs serving elderly users

Criteria for Stroke Rehabilitation	Criteria for Elderly Entertainment
- Adaptability to motor skill level - Meaningful tasks - Appropriate feedback - Therapy-Appropriate ROM - Focus diverted from exercise	- Appropriate cognitive challenge - Simple objective/interface - Motivational Feedback - Element of social activity - Appropriateness of genre - Creation of new learning following guidelines of experts - Sensitivity to decreased sensory acuity and slower responses

		Pong	Driver's SEAT	Whack-a-mouse	Tetris	Computer Chess	Trivial Pursuit
CRITERIA	Stroke Rehab	Adaptability to motor skill level	✓	✓	✓		
		Meaningful tasks	✓	✓			
		Appropriate feedback		✓			
		Therapy-appropriate ROM		✓			
		Focus diverted from exercise	✓	✓	✓	✓	✓
	Elderly Entertainment	Appropriate cognitive challenge			✓	✓	✓
		Simple objective/interface	✓	✓	✓	✓	✓
		Motivational Feedback	✓	✓	✓	✓	✓
		Element of social activity	✓			✓	✓
		Appropriateness of genre	✓	✓	✓	✓	✓
		Creation of new learning				✓	✓
		Sensitivity to decreased sensory acuity	✓	✓	✓	✓	✓
		Sensitivity to slower responses	✓	✓	✓	✓	✓

Envisioning collaborative virtual worlds 2010-2012



Virtual Life Demo Reel

Game-Based Virtual World Simulator Interfaces for immersive motorsports racing experiences



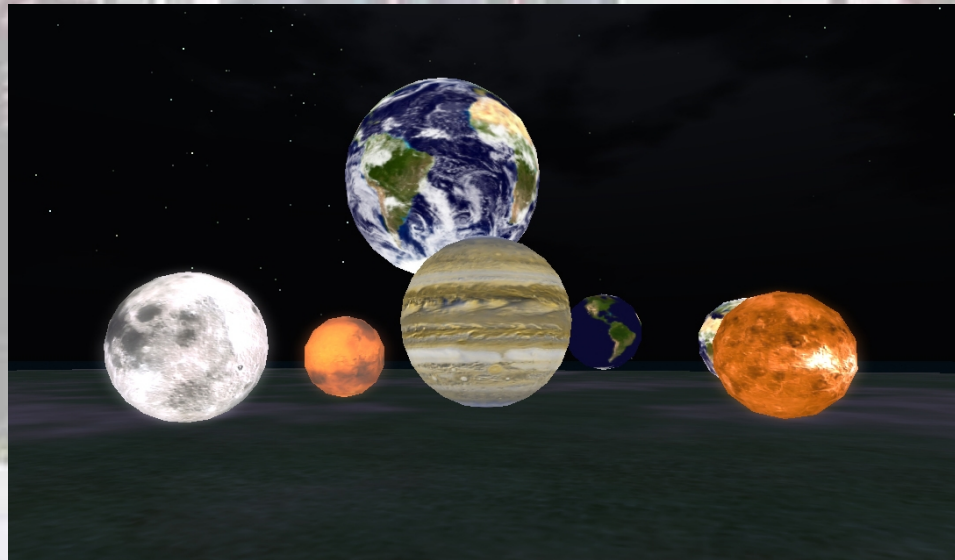
Game-based virtual world simulator you can actually drive in physical world! -- *OutRun* @ UCI



Future opportunities for games and virtual worlds

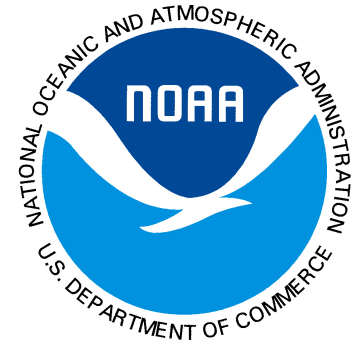
- Key challenges to address/overcome -- *scale and scope of:*
 - *Immersion*
 - *Verisimilitude*
 - *Within worlds*
 - *Spanning physical-virtual worlds*
 - *Co-participation and Collaborative work*
 - *Relocatability (telepresence)*
 - *Decentralized virtual organization*
- *New research center for Computer Games and Virtual Worlds at UCI*
 - <http://cgvw.ics.uci.edu>
 - *Funding from National Science Foundation #0808783, Digital Industry Promotion Agency (Daegu, South Korea), and others.*
 - *Want to come and play with us?*

Spherical Visualizations in Virtual Worlds



Spherical Displays in Virtual Worlds

We are experimenting with datasets from the
National Oceanic and Atmospheric Administration
(NOAA)'s Science on a Sphere (SOS) program.



Science On a Sphere Datasets



Science on a Sphere (SOS)



Science on a Sphere is a room sized, global display system that uses computers and video projectors to display planetary data onto a six foot diameter sphere, analogous to a giant animated globe.

Researchers at NOAA developed SOS as an educational tool to help illustrate Earth System science in a way that is both intuitive and captivating and to increase public understanding of the environment.

It is used as an instrument to enhance informal educational programs in science centers, universities, and museums across the world. It is currently installed at 47 sites worldwide.

<http://sos.noaa.gov/>

Discovery Science Center, Santa Ana, California



We are partnering with the Discovery Science Center, a local science museum in Santa Ana, which recently completed its own SOS installation.

Our goal is to bring the NOAA datasets to a broader audience using virtual worlds. We are looking to create a self-contained, open source solution that can easily be brought into the classroom.



SOS SphereCasting

An additional goal is to be able to receive SphereCasts from within a virtual world.



Viewing the first international SphereCast, live from COP15, the United Nations ClimateChange Conference in Copenhagen, Denmark from the Discovery Science Center in Santa Ana, CA.

What is SphereCasting?

A SphereCast is an SOS presentation done simultaneously at multiple sites by a single presenter, via the Internet. Many sites can receive the SphereCast, but only one site is the host. There are two components to a SphereCast: remote control of a presentation on an SOS system, and a live video (or audio) lecture that accompanies the SOS presentation.

The SphereCast is setup with a client-server architecture where the server is a specialized site (usually from NOAA headquarters) but any SOS installation can register to be a client/receiver of the SphereCast. Any commands the presenter issues to the SOS system at the host site are immediately replicated on all the SOS client systems. When the presenter loads a new dataset, that data is loaded on all the watching systems. When the presenter uses the remote control to start or stop animation, or orient the sphere, all the remote spheres behave identically.

All datasets must be pre-loaded in each client site. The streaming video component uses Apple Quicktime.

SOS Datasets/Playlists

- SOS Datasets include:

 - Images

 - Video

 - Animations

 - PIP (picture in picture overlays)

 - Audio

 - Slide presentations

 - Displayed on wall mounted displays, not on the SOS sphere

- SOS Playlists

 - Playlists organize and group together content for a presentation.

 - They can define the tilt and/or rotation of the sphere

 - They are simple text files that are read and interpreted by the main SOS application interface.

 - Playlists directly access the files in the datasets

OpenSim Platform

- We are experimenting with visualizing the SOS datasets in OpenSim.
- **OpenSim** is an open source server platform for hosting virtual worlds. It is compatible with the **Second Life** client.
- For the client, we are using the open source **Hippo** OpenSim viewer. It is compatible with the Second Life grid while taking advantage of OpenSim features and extensions.
- An OpenSim platform allows us to create a self-contained open source solution that can easily run on a portable computer.

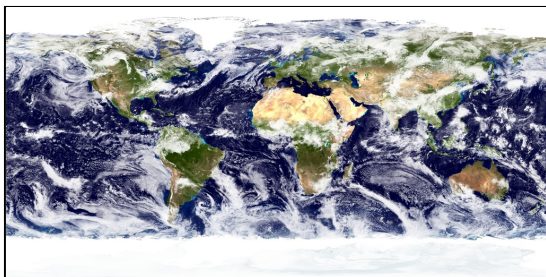
Goals

1. Display the SOS datasets in a virtual world using OpenSim.
2. Allow avatars to control which datasets are displayed.
3. Control the datasets from an external application.
4. Interpret SOS playlists and use them to control the datasets from an external app.

Datasets in OpenSim

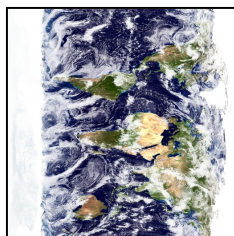
- Images

SOS images and video must have a 2:1 aspect ratio.

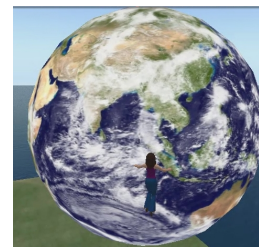


This “Blue Marble” image is an example image From the SOS dataset.

Images must be imported into OpenSim as textures that can be applied to the spherical prim object that implements the SOS sphere.



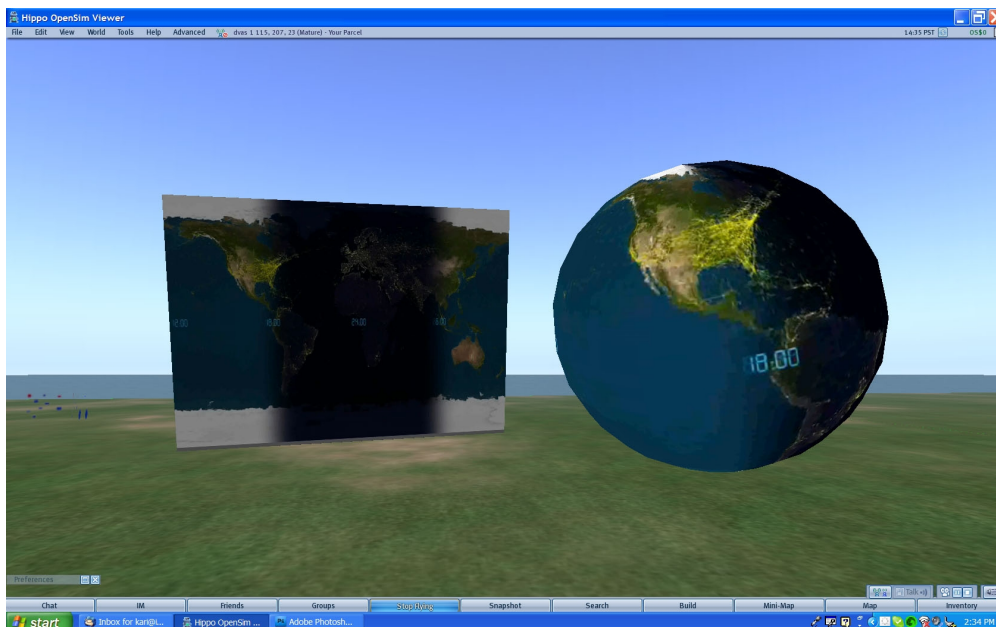
SOS images must be converted to 512X512 pixels and rotated before then can be imported as textures for the sphere (left). Note that the distortion created by scaling the image to a square format is counteracted when it is “stretched” around a spherical prim (right).



Datasets in OpenSim

- Video

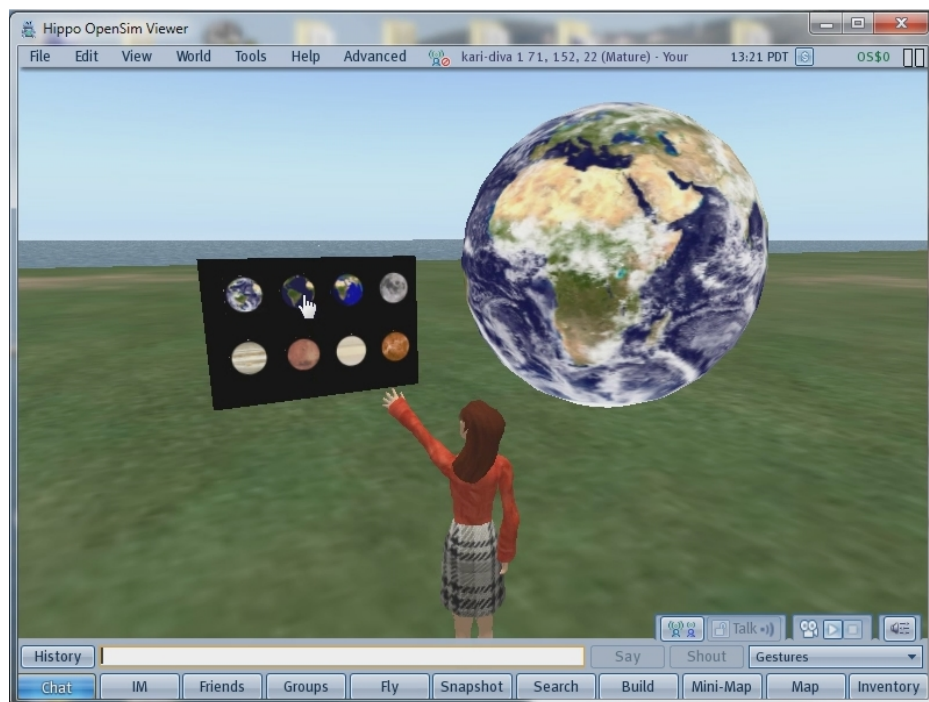
In OpenSim, mp4 video can be streamed from an external URL onto any object covered with a predefined “media texture”.



Here a video showing international aviation flight paths relative to time is being streamed simultaneously to both a rectangular and a spherical prim. The video is adjusted to wrap seamlessly around the sphere.

Controlling Datasets

- Using an avatar



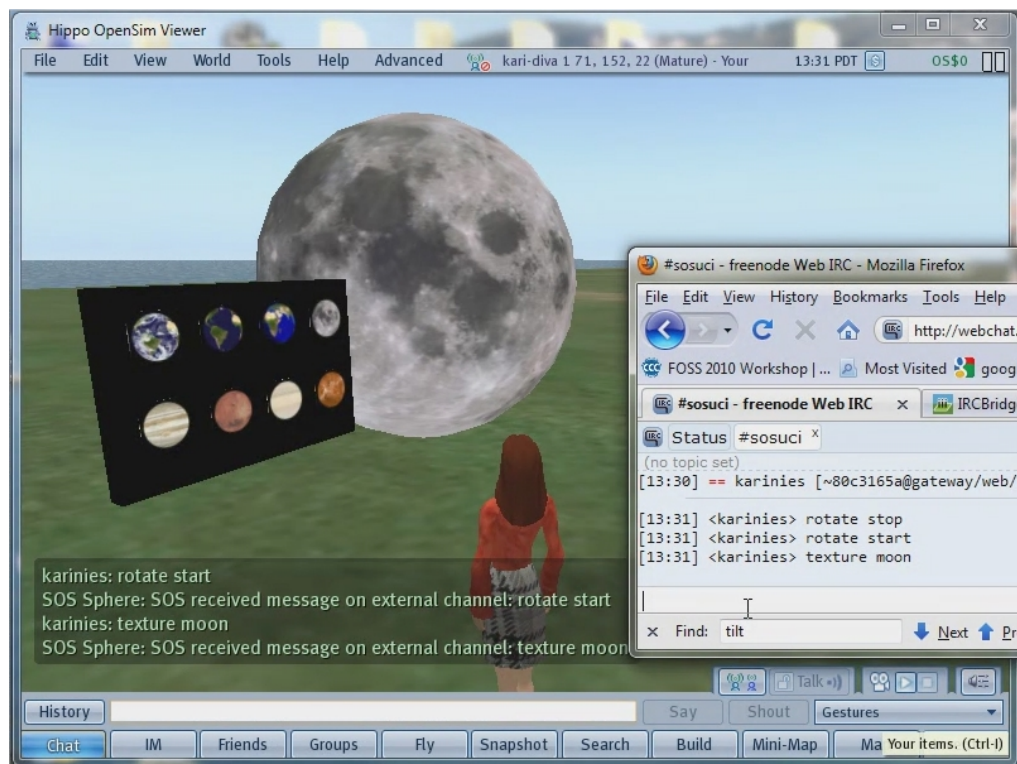
Here an avatar controls which dataset is displayed by “touching” a kiosk with a variety of planet display options.

The avatar can also “touch” the sphere to start and stop its rotation.

The touch events are detected and handled via scripts (written in LSL-Linden Scripting Language). Communication between objects is via internal chat channels.

Controlling Datasets

- From an external chat channel

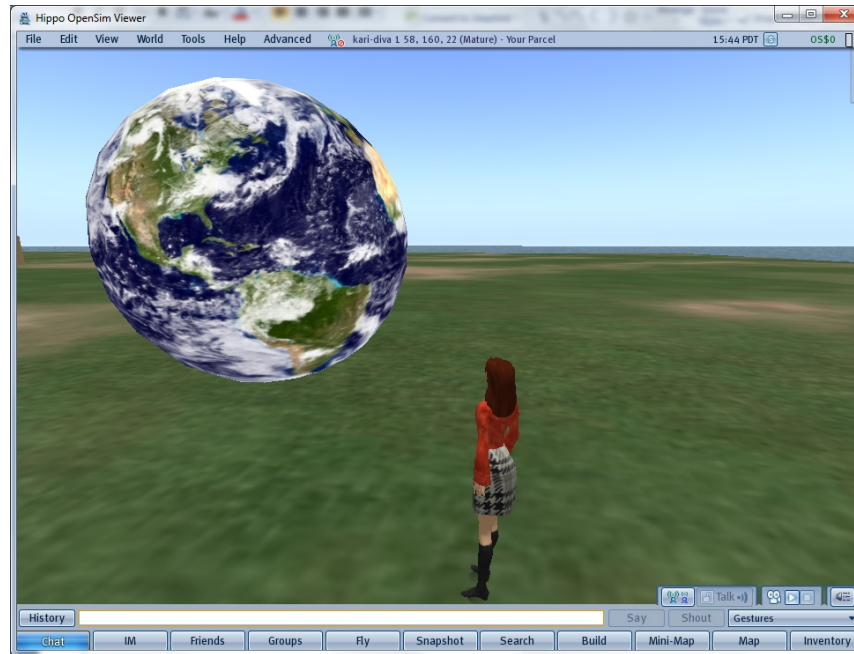


Here the client has been configured to listen to an external chat channel on freenode.net.

The user can send commands from the external channel. Here a user is controlling rotation and changing the sphere texture remotely.

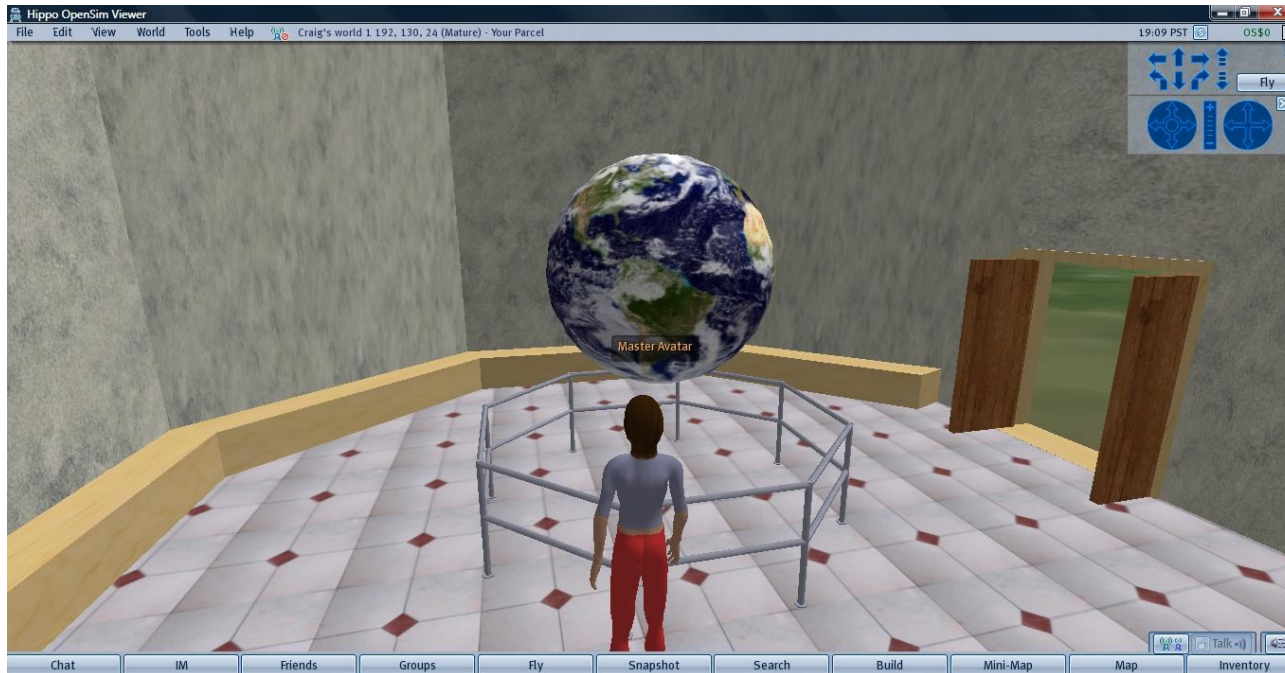
Tilt

- Scripts now support displaying the sphere at a specified x-axis tilt.



The SOS datasets display earth images at a 23.5 deg. tilt on the x-axis.

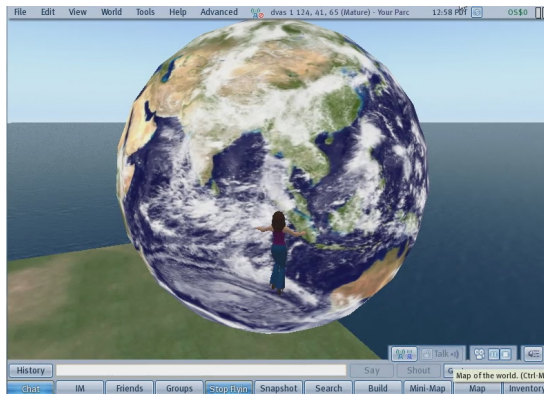
Modeling an SOS Installation



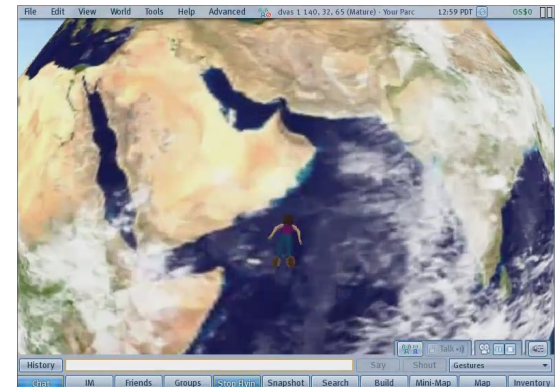
We are in the early stages of modeling the Discovery Science Center's SOS Installation in OpenSim.

More Spherical Visualizations

- Hollowed Sphere



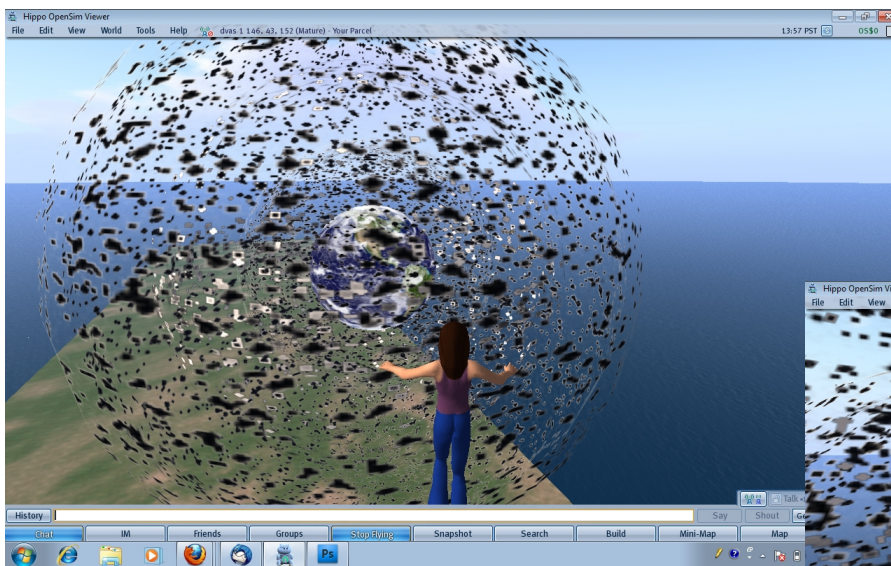
Here an avatar is shown entering a hollow sphere that contains a smaller sphere in it's core. Both are textured with SOS data images.



In this view the Avatar has entered the interior of the larger sphere and is looking outward. Note that the texture image is visible for the spheres interior and also that the image is a reversed from the inside looking out.

More Spherical Visualizations

- Space debris visualization using hollowed spheres



We've been exploring visualizations of space debris in virtual worlds.

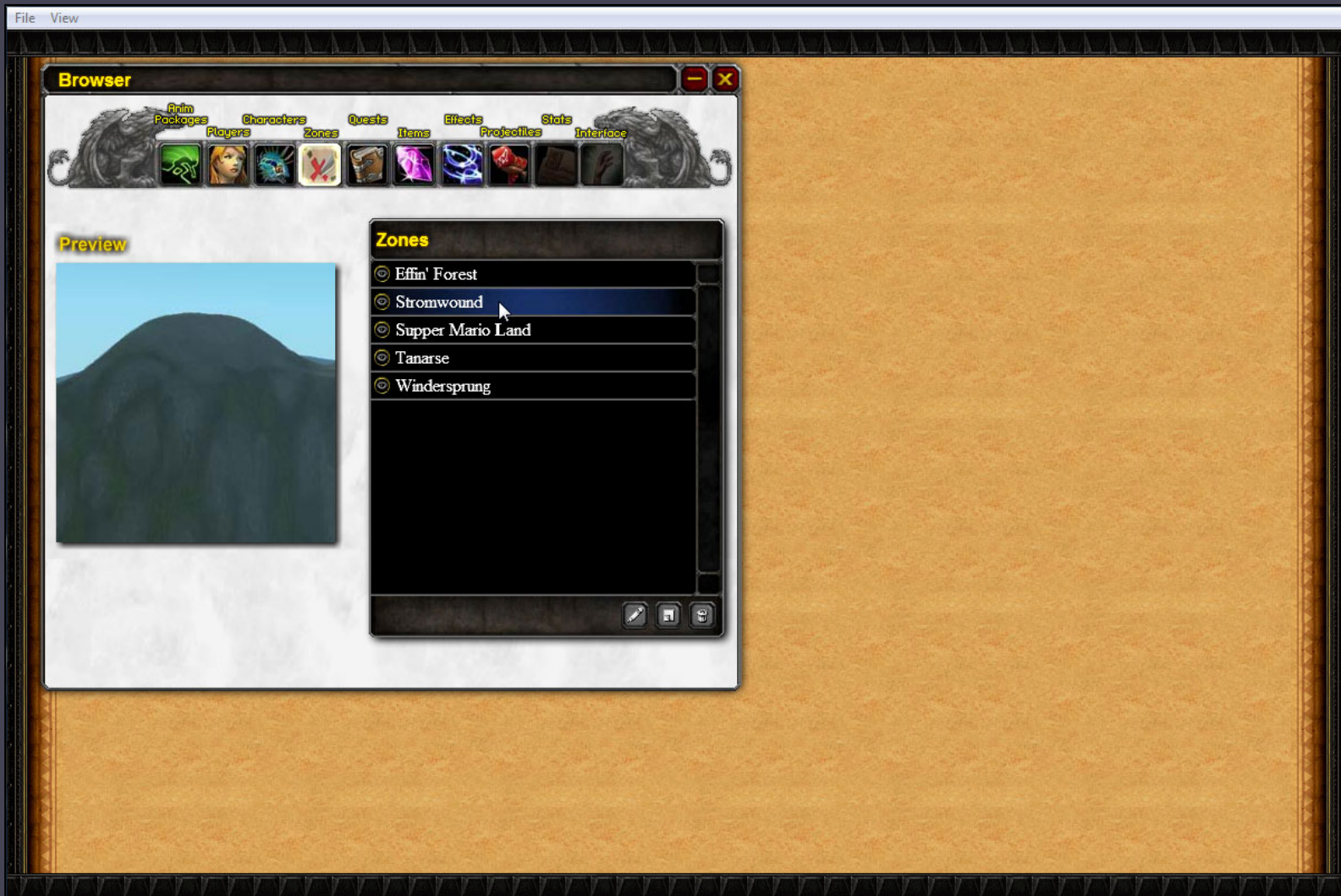
Here we've simulated a visualization of debris in the earth's atmosphere. Surrounding the earth textured sphere in the center are three concentric hollowed spheres with transparent textures.



"!" - Game Modding and Development Kit

(A Work Nearly Done)
'08-'10

Asset Browser



Zoom Image

- WoW inspired side-scrolling action RPG game modding and development environment
- Built in Flash using Adobe Air
- Opens to a base project canvas that manages multiple game editing windows
- Asset Browser previews all game assets and launches editors

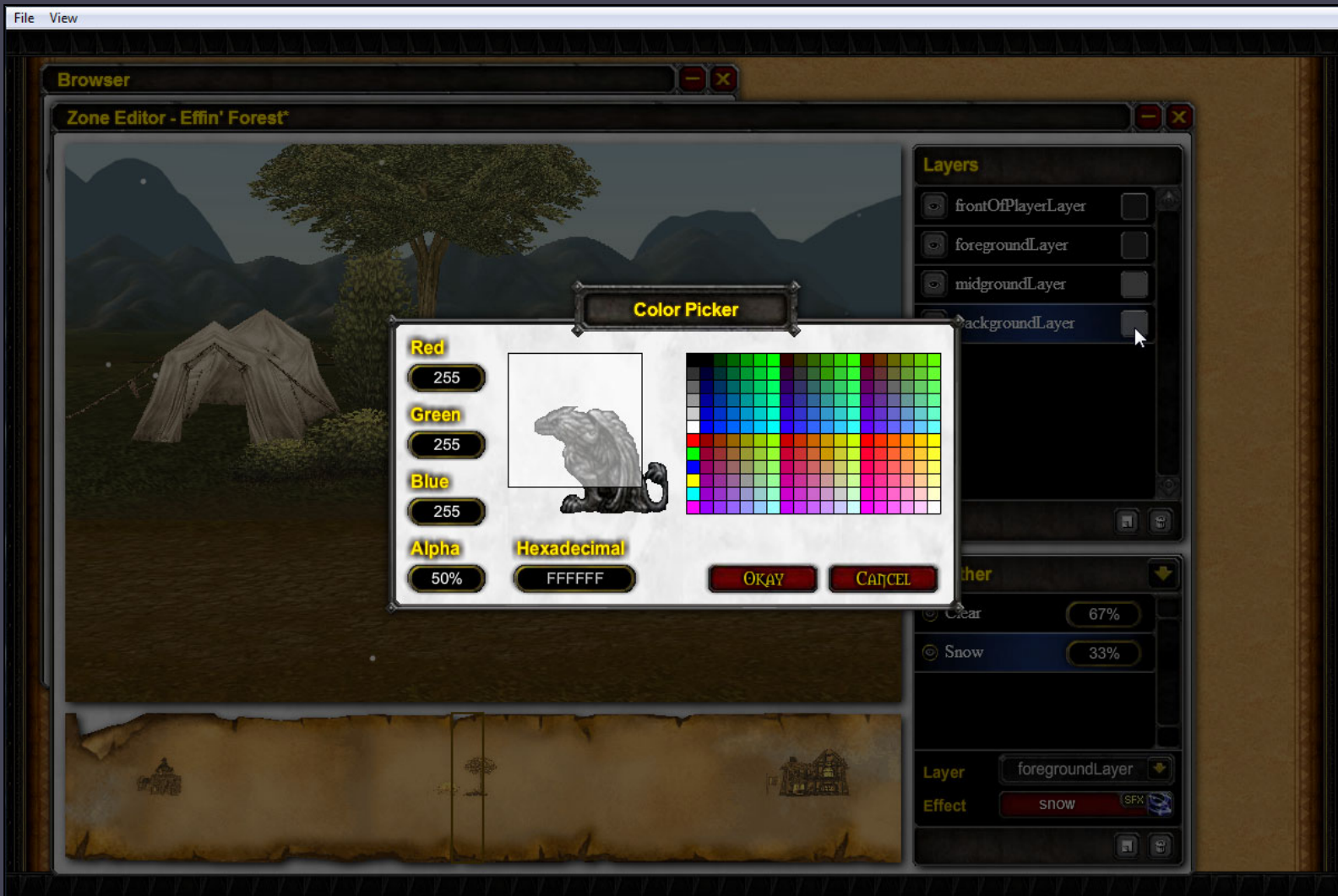
Level Editor



Zoom Image

- Photoshop style editing mode
- Import any number of layers to place objects
- Adjust scroll speed of layers independently for parallax scrolling
- Create custom weather conditions with a percent chance of occurring
- Click and drag to scroll zone and place objects, characters, spawn points, spawn conditions, etc
- Minimap supports placement and manipulation of objects as well

Color Picker



Zoom Image

- Assign each layer unique graphics and color tinting
- Assign key-framed time of day tinting changes based on a 24 hour clock that automatically interpolates color values

Quest Editor



Zoom Image

- Supports creation of multiple quest types - collect, kill, escort, object initiated, chained, etc
- Establish requirements for access, rewards, items needed for completion, and more
- Define relationships to other quests
- Create quest text for introduction, objectives, progress, completion

Item Editor



Zoom Image

- Create items associated with quests and random drops
- Set descriptive "flavor text"
- Assign quality, type, subtype, level requirements, cost, bonuses, ability cost/effects

Ability Setting



Zoom Image

- **Set item abilities such as type, whether item effects stack, stack limits, duration**
- **Link associated graphics**
- **Create descriptive text summarizing abilities for player**

Requirements Setting



Zoom Image

- Establish hierarchies of requirements that must be met prior to items and quests becoming available

Character Editor



Zoom Image

- Create custom characters
- Choose attributes and behaviors based on pre-defined templates or create your own
- Establish identity markers (faction, profession, race, class, and anything else you want)
- Attach pre-set animations or create your own
- Create character tinting effects based on state
- Determine default equipped gear
- Author chat dialogs
- Define complex sets of behaviors
- Set loot tables
- Decide if character is a quest giver, and which quests are available if so
- Preview the settings

Projectile Editor



Zoom Image

- Choose from a seed particle list to build more complex projectile effects
- Set properties for projectiles (name, duration, damage, trail effects, movement patterns, etc)
- Preview the settings

Particle Effects Editor



Zoom Image

- Choose from a seed particle list to build more complex particle effects
- Associate these effects to projectiles or other items and conditions
- Set properties for particles (whether static or dynamic, behavior, appearance, amount, duration, etc)
- Link custom made sounds to particle effects
- Preview the settings

Animation Editor



Zoom Image

- Browse and link a series of image frames to custom character animations
- Associate animations with behavior states (idle, jump, attack, stun, death, cast, etc)
- Define which frames get used
- Set bounding boxes to establish impact areas
- Preview the settings

Character Action Editor



Zoom Image

- Define specific actions to use to control custom NPC behaviors
- Tie actions to custom animations and specify primary and secondary use types
- Adjust interaction settings and effects
- Specify movement speed and angle, whether action is interruptible, sounds, and particle effects
- Set impact effects including damage multipliers, range, force, angle, and special effects

Character Behavior Editor - Flow View



Zoom Image

- Create complex NPC behaviors and scripted events
- View behaviors in "Flow" mode (diagram) or "Full" mode (code)
- Define custom states for NPCs to switch to based on action conditions

Character Behavior Editor - Full View



Zoom Image

- Set NPC faction based on specific states
- Set NPC auras (ground color upon selection) independant of faction
- Link a variety of predefined conditions to code blocks
- Easy enough for non-programmers to use

Character Dialog Editor



Zoom Image

- Define complex NPC and player dialog trees
- Trigger dialogs based on custom state lists
- Allows NPCs to "make" player character speak/respond
- Establish percent likelihood of dialog to occur

Character Quest Handling Editor



Zoom Image

- Determine which quests get handled by particular NPCs
- Define whether NPC gives and /or finishes a specific quest, and /or serves as a progress checker
- Create custom greetings for quest-handling NPCs

Character Equipment Editor



Zoom Image

- Link items to characters
- Create custom abilities for characters (similar to the WoW spellbook)
- View the combined stats of characters as equipped

Player Editor



Zoom Image

- Define characters to use as main player character (up to 4)
- Create custom identity markers that can be tag-referenced to display in quest text to create a personalized feel
- Define a special login animation for playable characters to display on startup
- Link unique background graphics and icons to playable character login screen
- Select unique starting for each playable character
- Set custom player capabilities such as run, jump, and attack and dodge velocity / angle

PORTAL

OutRun: Pervasive Games and Designing the De-Simulation of Eight-Bit Driving

Garnet Hertz

Center for Computer Games and Virtual Worlds
Institute for Software Research
University of California, Irvine
209 Information and Computer Science 2
Irvine, CA 92697, USA

ghertz@uci.edu (<http://conceptlab.com/>)

ABSTRACT

This paper outlines the development process of a mixed reality video game prototype that combines a classic arcade driving game with a real world vehicle. In this project the user, or player, maneuvers the car-shaped arcade cabinet through actual physical space using a screen as a navigational guide which renders the real world in the style of an 8-bit video game. This case study is presented as a “pervasive game”: an attempt to disrupt the everyday by highlighting and inverting conventional behavior through humor and paradox.

Categories and Subject Descriptors

H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems – *artificial, augmented and virtual realities*; H.5.m [Simulation and Modeling]: Types of Simulation – *gaming*; J.5 [Arts and Humanities]: Fine Arts, Performing Arts; H.5.m [Information Interfaces and Presentation]: Miscellaneous.

General Terms

Design, Experimentation, Human Factors, Theory.

Keywords

Game design, game innovation, mixed reality, augmented reality, alternate reality, pervasive gaming, live action role playing, electronic art, experimental interfaces, prototyping.

1. INTRODUCTION

The OutRun project is a game and media art project that explores the overlap between the physical world and game environments. OutRun explores the act of driving a vehicle and the interstitial space between everyday life (driving an automobile) and a simulation of it (playing a driving video game) by combining the real world and OutRun, an eight-bit arcade driving game released by Sega in 1986. This project features two main components:

1. **Cabinet-Car:** A car-shaped sit-down arcade cabinet from Sega's OutRun is converted into a small car that can actually drive. This is done by modifying an existing fiberglass and wood cabinet with motors, wheels and drivetrain components from an electric golf cart. The original arcade cabinet is modeled after a 1984 Ferrari Testarossa. This customized cabinet-car will use the existing videogame controls (steering wheel, acceleration pedal,

brake) to control the vehicle. It is expected that the maximum speed of the car will be no more than 20 miles (32 kilometers) per hour. See Figure 1 for a mockup diagram of the completed cabinet-car.

2. **Custom Augmented Reality Software:** The screen, which is positioned in front of the driver, renders the real world in the style of the 1986 video game OutRun. This is done through custom-built computer vision software and GPS sensors that calculate the location of the experimental vehicle and display a street-level view rendered in the style of the vintage video game. In other words, the driver only sees an eight-bit-style game rendered in their “windshield,” which appears as if they are playing the 1986 videogame. Accelerating or turning the car-cabinet in the real world will proportionally change the display. Although the screen will mimic the real world around it, it is expected that the augmented display and the real world will not match perfectly.



Figure 1. The proposed OutRun cabinet-car.

This project is motivated by the following concepts:

1. **The De-Simulation of Driving** - This project de-simulates the driving component of a videogame. Driving game simulations strive to be increasingly realistic, but this realism is usually focused on graphical representations. Instead, this system pursues “real” driving through a videogame as its primary goal.

2. **GPS Navigation & Mixed Reality Parallax** - Driving in a real automobile with a GPS navigation system can be game-like. This project explores the consequences of only using only a computer

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DRAFT: NOT FOR CITATION OR DISTRIBUTION

model of the world as a navigation tool for driving. The windshield of this project's vehicle only shows a re-rendered simulation of the immediate environment, and as a result, driving it in the real world is often difficult or dangerous. As a result, this project explores and investigates how augmented reality and GPS data differs from the physical world, and what happens when an augmentation of reality envelops and obfuscates reality.

2. CONTEXTS IN GAME CULTURE

The OutRun project shares similarities to several existing genres of game design, namely pervasive games (including live-action role-playing games, alternate reality games, and big games) and driving simulators.

Pervasive games “move beyond the traditional computer interfaces and into the physical world to occupy time and place on a human scale.”[Falk, Jennica; Davenport, Glorianna (2004). "Live Role-Playing Games: Implications for Pervasive Gaming" (PDF). Entertainment Computing – ICEC 2004. Lecture Notes in Computer Science. 3166. Springer Berlin / Heidelberg. page 127.] As Falk and Davenport describe, pervasive games blur lines of demarcation between games and life: players are mixed with game characters, the real world is overlapped with game worlds, and game artifacts are combined with real world objects. Contemporary pervasive games consist of at least three subgenres: live action role-playing games, alternate reality games and big games.

Live action role-playing (LARP) games are pervasive role-playing games where individuals physically act out their characters' actions, typically in a fictional setting that is represented by the real world. Players embody their character roles through costumes and physical actions, maintaining a clear sense of being in-character or out-of-character [Tychsen et al., Live Action Role Playing Games, page 255]. Alternate Reality games, on the other hand, are pervasive games that attempt to delete the line between being in-character and out-of-character: “they do everything in their power to erase game boundaries – physical, temporal and social – and to obscure the metacommunications that might otherwise announce, 'This is play.'”[Jane McGonigal, A Real Little Game: The Performance of Belief in Pervasive Play]. Instead of fictional character play, alternate reality games have an aesthetic of trying to not look like either a game or a fantasy. Inspiration for alternate reality games is primarily driven by an extension of gaming environments through mobile network technologies and ubiquitous computing. [Jane McGonigal, A Real Little Game: The Performance of Belief in Pervasive Play].

Within the category of pervasive games, the OutRun project perhaps has the most similarity with some styles of “Big Games,” especially the Pac-Manhattan project led by Frank Lantz at NYU's Interactive Telecommunications graduate program in 2004 (Figure 2). Pac-Manhattan is a large-scale urban game that utilized the New York City grid to recreate the 1980's video game Pac-Man. A player dressed as Pac-Man runs around an area of Manhattan while attempting to collect all of the virtual “dots” that ran the length of the streets; four players dressed as the ghosts Inky, Blinky, Pinky and Clyde attempted to catch Pac-Man before all of the dots were collected [Pac-Manhattan, <http://pacmanhattan.com/about.php>, 2004].



Figure 2. Pac-Manhattan, a “big” game developed in 2004 where individuals play a Pac-Man inspired physical game through the streets of Manhattan.

Like the OutRun project, Pac-Manhattan takes an existing eight-bit game and explores what happens when it is mapped into the larger “real world” of buildings, streets, and cities. In comparison to LARP or Alternate Reality games, Big Games like Pac-Manhattan have a simple premise: take an existing video game and explore what happens when it is scaled up into the real world. OutRun does this, but also does the inverse: it attempts to take the real world and translate it into a “small game” on a computer screen.



Figure 3. The F1Showcar, a full size motion simulator computer game car system.

The OutRun project also borrows from non-pervasive game systems in its concept. The contemporary racing simulator video game community, for example, has developed a substantial selection of customized computer furniture and hardware peripherals to augment its genre of game play. Basic systems include foot pedal and steering wheel peripherals, while more involved configurations include full scale car cockpits and hydraulic motion tables that simulate the physics of real-world driving (Figure 3) [F1 ShowCar, <http://f1showcar.com/>].

The OutRun project extends this pursuit of immersive driving simulation by reconnecting the reproduction with its original source: actually driving. In the process, OutRun borrows from pervasive gaming by using the real world as its game space.

The end result explores the relatively new phenomenon of driving an automobile and simultaneously looking at a computer screen. Contemporary driving increasingly involves computing: using a GPS navigation system, in-dash entertainment, or text messaging while driving are progressively part of real-world driving culture. Although automobile GPS navigation is not considered a game, malfunctioning wayfinding algorithms and inaccuracies in street-level data can produce a humorous, frustrating or dangerous “pervasive game.” These overlaps have serious consequences and deserve to be explored: in 2008, the British newspaper *The Mirror* estimated that automobile GPS navigation systems have caused 300,000 accidents in the UK [The Mirror, SatNav danger revealed: Navigation device blamed for causing 300,000 crashes, Tanith Carey, 21/07/2008]. OutRun is not proposing a solution to precarious uses of computers in vehicles, but intentionally magnifying and embracing the problem. In the process, the system strives to function in the historical role of a trickster: to disrupt the everyday by highlighting and inverting conventional behavior through humor and paradox. [Hertz, Un-Simulations, Tricksters and Radical Thought, Blackflash Magazine, 2009] OutRun extends simulation and perverts it at the same time: it is like a fantasy taken to its extreme and gone humorously wrong. Being a clown doesn't accomplish much, but trying to intelligently invert social conventions and assumptions can constructively and artistically rewire our understandings of who we are.

3. PROJECT DESIGN AND DEVELOPMENT PROCESS

The OutRun system began as a project proposal in January 2009. Active development started in June 2009 after five undergraduate students at UC Irvine were recruited as interns from the course “Computer Games as Art, Culture and Technology” [Losh, DAC09, 2009]. I oversaw development production, and the team was split up into the following general areas – Hardware: Matt Wong and Erik Olson; Game and Visualization Software: Chris Guevara; Media Assets: David Dinh; and Documentation: Richard Vu. The Laboratory for Ubiquitous Computing and Interaction at UC Irvine provided intern office space and software licenses, while the Arts Computation Engineering program at UCI provided large format studio space for the physical construction of the cabinet-car.

3.1 Software Design and Development Process



Figure 4. A screenshot from the first level of the original 1986 video game OutRun.

The original OutRun game featured a pixelated 1984 Ferrari Testarossa that the user controlled down winding roads through a number of different landscapes. For the purposes of this project, only the first level will be used as a source of inspiration: it is an idealized version of a California-esque beach community with palm trees, windsurfers and surf shops – plus a six lane freeway barreling through it all (Figure 4).

Software development of the project to date has focused on building a software system that is able to render the real world as it would be drawn in the original OutRun game. In other words, the software's key task was to draw the world like an eight-bit driving videogame: if a road actually appears in front of you that curves to the left, the system would draw an eight-bit style road on the screen that curved to the left.

Development of this software has taken two different paths: 1. Using computer vision to detect objects in the real world, and 2. Using location-aware technologies like global positioning systems (GPS) to position the user in a virtual world.

3.1.1 Computer Vision Development Path

Development of the OutRun software began with the development of a realtime computer vision system using Max/MSP/Jitter 5, an interactive graphical programming environment for music, audio, and media [Cycling '74, Max/MSP/Jitter, <http://cycling74.com/>]. Jitter – an architecture optimized for use with video – was used to take video data and store it in a two dimensional array. Cvjit, a collection of of Max/MSP/Jitter computer vision tools by Jean-Marc Pelletier, was used for its ability to detect line edges in video data. Along with pre-defined Max/MSP/Jitter objects and a line detection tool in cv.jit, we created our own computer vision objects in Java to tackle what we saw as our main problem: to detect the shape of roads.



Figure 5. Driving video processed using Jean-Marc Pelletier's cv.jit.lines, a Max/MSP/Jitter tool for detecting line edges in video data.

Cvjit detected numerous lines in a scene (Figure 5) and to focus on roads, we developed a number of custom filtering algorithms to remove lines that we thought weren't road-like. Then, by calculating the convergence point of the road-like lines, we tried to determine the general direction of the road (Figure 6).



Figure 6. Driving video processed with cv.jit.lines and custom filtering modules. The dot in the center of the image shows the calculated direction of the road, which is the average convergence point of filtered lines.

This approach has been useful in programmatically determining the direction of real-world roads, but is far from perfect. The current system is only set to detect a single road endpoint, does not calculate road width, and gets confused by shadows, crosswalks and other variables.

Despite these problems, the calculated endpoint produces interesting results when rendered in an eight-bit driving video game style. To do this, we constructed a simple Adobe Flash-based driving game that was controllable by our Max/MSP/Jitter vision software. We used Antony Dzeryn's Retro Racer as a starting point for this development, a Flash-based open source driving game that was similar in style to the original OutRun game. By modifying the ActionScript and replacing media assets, we modified Retro Racer to look more like OutRun and enabled communication between Max/MSP/Jitter and Flash (Figure 7).



Figure 7. Custom Flash-based game developed in the style of OutRun, with road endpoint controllable by computer vision data in Max/MSP/Jitter.

3.1.2 Locative / GPS Development Path

The second development path that is being pursued for this project is through location-aware technologies like global positioning systems (GPS) to position the user in an eight-bit style virtual world. This development path has been pursued with undergraduate students Matt Shigekawa, Jesse Joseph and Mike Tang starting in September 2009.

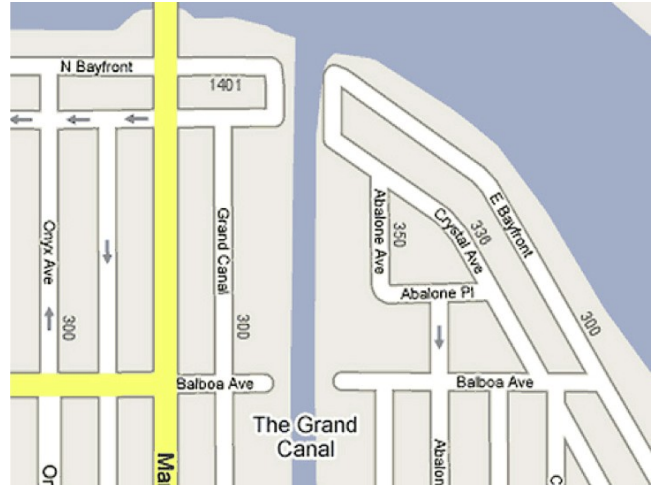


Figure 8. A standard Google Map view of Balboa Island, California that shows street data and land outlines.



Figure 9. Street data and land outlines of Balboa Island drawn in an eight-bit videogame style.

This approach takes data from the physical world – like street data, land shapes and building placement – and builds a pre-rendered three dimensional virtual world that can be navigated by using a real time GPS sensor that controls a handheld display.

The three dimensional virtual world was built to be displayed in Google Earth, and was constructed in Keyhole Markup Language (KML), an XML-based language schema for expressing geographic annotation and visualization on two-dimensional maps and three-dimensional Earth browsers [Google, KML – Google Code, <http://code.google.com/apis/kml/>]. To simplify development and testing of this approach, we focused on small geographic areas: Balboa Island and the UC Irvine Campus in

Orange County, California. Ground maps for this eight-bit-styled environment were built non-programmatically from street and earth data (Figures 8 and 9), and objects – like trees, buildings, automobiles and people – were taken from the original OutRun game. The world was assembled using Google SketchUp, a 3D modeling program that facilitates the placement of models in Google Earth (Figures 10 and 11) [Google SketchUp, <http://www.sketchup.com>].

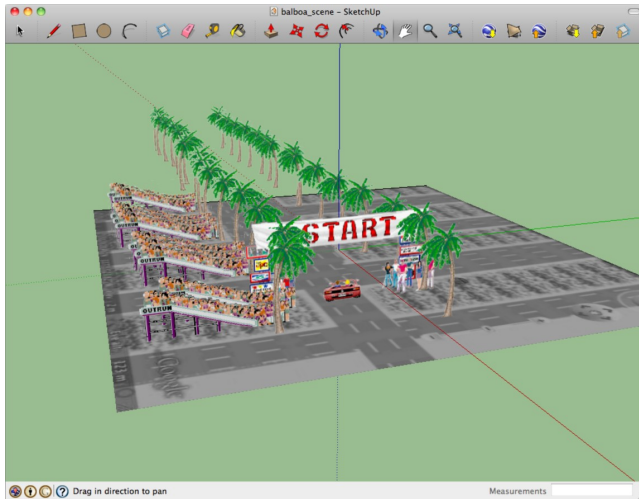


Figure 10. Two dimensional media assets being assembled using Google SketchUp, recreating the start sequence of the original OutRun game on Park Avenue, Balboa Island.

After assembling scenes on Balboa Island and the UC Irvine Campus the locations were physically navigated with a Q1UP-XP Samsung UMPC Tablet and a Nokia LD-1W Wireless GPS Module, with position data fed from the GPS module to Google Earth by GooPs Pro, a Google Earth GPS Tracking and Navigation tool.



Figure 11. View down Park Avenue of Balboa Island in Google Earth with placed OutRun-style media assets.

Real-world testing of this locative/GPS system showed that the Nokia LD-1W Wireless GPS module had a resolution of approximately ten to fifteen meters and a location refresh rate of

about three seconds. These factors made real-time interaction with the virtual world not as immersive as we would have liked: in Balboa Island tests, the lack of GPS resolution positioned us as driving through a row of trees and the position refresh rate produced a lurching/hopping sensation as you moved down the street, at least at the perspective of the original video game. A locative development path will likely require a high resolution GPS receiver (like a Navcom SF-2050G with an accuracy of 10cm) and tweening between location waypoints.

The sensation of speed was also considerably different from the original game: driving a real-world car at 40 kilometers per hour through the eight-bit virtual world, for example, seemed painfully slow in comparison to the 200+ km/h average speed of the original game car. In our mixed reality GPS prototype it felt as if you were crawling through the game world, highlighting that in everyday driving one normally doesn't race a car down residential streets. This gap between the expectations of the game world and action in the real world could have some interesting implications for gameplay: it could either encourage players to drive recklessly in the real world to recreate the original eight-bit game experience, or highlight how slow and mundane everyday real-world driving is in comparison to a driving game.

3.2 Cabinet-Car Design and Development Process

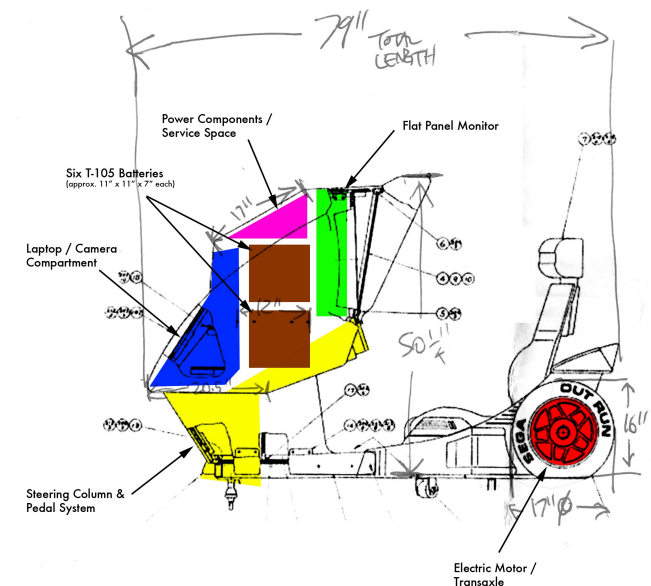


Figure 12. Proposed Cabinet-Car Component Layout, using side profile of video game cabinet as a reference.

The cabinet-car is the conglomeration of two components: a sit-down video game arcade cabinet and an electric golf cart (Figure 12). The arcade cabinet chosen was the “deluxe” OutRun configuration offered by Sega, weighing with 350 kg (770 lbs) with hydraulic side-to-side motion, a large fiberglass driving cockpit and a 25 inch CRT monitor. Our cabinet was purchased from a private party in Maine (USA) in July 2009 (Figure 13).



Figure 13. Original OutRun "Deluxe" arcade game cabinet (background) and 1959 Turfrider Mark IV golf cart (foreground).

In August 2009 a three-wheeled golf cart was purchased as a drivetrain for the system: a 1959 Turfrider Mark IV (Figure 13). The Mark IV golf cart was selected due to its three wheel configuration, unique "googie" space-age styling and similar form factor to the OutRun cabinet. However, after testing the system it lacked efficiency, power, stability and the tricycle steering mechanism would be difficult to interface with the arcade steering wheel. As a result, the classic golf cart was abandoned in favor of a four-wheeled 2007 Ez-Go RXV, which grants at the Institute for Software Research supported since Fall 2009. The Ez-Go RXV electric golf cart provides a more functional drivetrain, with greater reliability, power, battery duration and serviceability to make the make the system more viable as a robust exhibit/vehicle device.

4. CONCLUSION: PERVERSIVE GAMING

In 2003 Jane McGonigal humorously raised the idea that the combination of pervasive and immersive play results in "pervasive" gaming: where "players are prone to falling for the games' dissimulative rhetoric [...] They wind up believing in their play too much for their own good." [Jane McGonigal, "A Real Little Game: The Performance of Belief in Pervasive Play." Digital Games Research Association (DiGRA) "Level Up" Conference Proceedings. November 2003.] Through the OutRun project, I'd like to return to McGonigal's concept of pervasive gaming and extend it.

McGonigal sees that in pervasive and immersive game situations gamers maximize their play experience by *performing* the role of believing that they are able to permeate the game-reality boundary. In other words, players don't actually believe that real life and games are the same thing, but their enjoyment flows from pretending that the two worlds are able to be intermingled.

McGonigal describes this gameplay as a combination of the "Pinnocchio effect" – the desire for a game to be transformed into real life – and conversely, the desire for everyday life to be transformed into a "real little game." McGonigal focuses on two examples in her paper: the 2001 immersive game known as the Beast, and the Go Game, an ongoing urban superhero game [ibid].

The OutRun project is designed to fail as a device that will seamlessly permeate the boundary of real life and a game. In other words, it is intentionally built with faulty logic at its core: attempting to map a 1986 video game into the real world likely will never result in a seamless transition between game and life. When the game is extended beyond its normal constraints as a videogame, it results in a malfunctioning: confusion, collisions or accidents. However, this is part of the fun of the OutRun project.

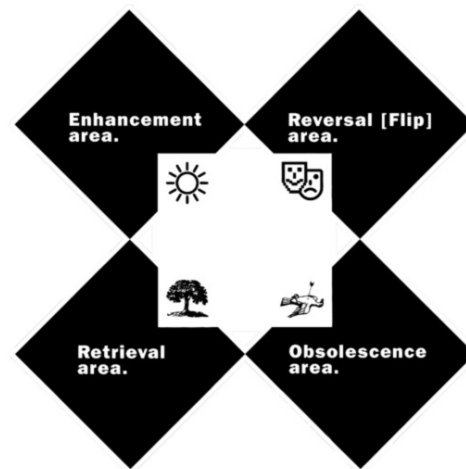


Figure 14. McLuhan's Tetrad diagram, which proposes how communication and representation technologies have the potential to change in four distinct ways: obsolescence, retrieval, enhancement, and reversal.

A useful framework for visualizing and rethinking this dynamic includes Marshall and Eric McLuhan's Laws of Media: The New Science (1989), which proposes that media technologies have the potential to change in four distinct ways [Marshall and Eric McLuhan, Laws of Media: The New Science, 1989]. A poetic four-region model is presented to envision the characteristics of obsolescence, retrieval, enhancement, and reversal (Figure 14). The reversal, in McLuhan's eyes, occurs when something is pushed to its limits.

The main point of relevance to this discussion on pervasive and immersive games is that when mediating technologies are pushed beyond their ordinary limits, they can reverse or flip in their intent or use. The McLuhan tetrad diagram gives a visual graph to think about how game projects may simultaneously amplify, invert, revive, and subsume – they swirl around and don't simply proceed in a straight line. When a simulation is taken beyond its role as a safe fantasy and pushed to envelop and take over reality, it becomes perverted. Problematising a binary virtual/physical view of the world, for example, can be done by taking virtuality to an extreme. By taking a belief or simulation too far, it stops being a comforting diversion and flips into an absurdity, an obsession or a dark dream [Hertz, Un-Simulations, Tricksters and Radical Thought, Blackflash Magazine, 2009]. In the process, beliefs are "perverted" [Rafael Lozano-Hemmer, Perverting Technological Correctness, Leonardo 29:1, 1996].

OutRun does not propose a solution to a problem: it intentionally magnifies and embraces an impossible gap between simulation and reality. It does not see a *line* between game and real life, but rather a pleasurable impossible *area*: an uncanny valley between the familiar and the almost-familiar [Masahiro Mori, 1970]. Exploring this “perverted” and uncanny valley by over-extending simulations is a useful approach for game design. Intentional misuse of technology, incorporating physical pain, questioning cultural concepts of progress, creating intentionally useless devices and bending cultural stereotypes are all useful approaches to embrace in developing the uncanny [Rafael Lozano-Hemmer, *Perverting Technological Correctness*, Leonardo 29:1, 1996].

In this perversion, the game strives to function to break the “natural” rules to highlight and invert common knowledge. To build perverse games is to explore the historical role of a trickster: to disrupt the everyday by highlighting and inverting conventional behavior through humor and paradox. In the process, the contractual magic circle of engagement is extended spatially, temporally, and socially [Markus Montola, *Games and Pervasive Games*, Page 7].

5. ACKNOWLEDGMENTS

Thanks to the development efforts of Walt Scacchi, Professor Jong Weon Lee, Jeremy Bailey, Matt Wong, Erik Olson, Chris Guevara, David Dinh, Matt Shigekawa, Jesse Joseph, and Mike Tang.

The OutRun project has been supported by the Laboratory for Ubiquitous Computing and Interaction at UC Irvine, Gillian Hayes, Christina Lopes, the Arts Computation Engineering Program at UC Irvine, and is currently supported by the Center for Computer Games and Virtual Worlds and the Institute for Software Research at UC Irvine. Support for this research is provided through grants from the National Science Foundation #0808783; Digital Industry Promotion Agency, Global Research and Development Collaboration Center, Daegu, South Korea. No endorsement implied.

6. REFERENCES

- [1] Cycling '74, Max/MSP/Jitter, <http://cycling74.com/>.
- [2] Google, KML – Google Code, <http://code.google.com/apis/kml/>.
- [3] Google SketchUp, <http://www.sketchup.com>.
- [4] Garnet Hertz, *Un-Simulations, Tricksters and Radical Thought*, Blackflash Magazine, 2009.
- [5] Elizabeth Losh, *Hybridizing Learning, Performing Interdisciplinarity: Teaching Digitally in a Posthuman Age*, Digital Arts and Culture 2009.
- [6] Jenna Falk and Glorianna Davenport (2004). "Live Role-Playing Games: Implications for Pervasive Gaming" (PDF). *Entertainment Computing – ICEC 2004. Lecture Notes in Computer Science*. 3166. Springer Berlin / Heidelberg. Page 127.
- [7] F1 ShowCar, <http://f1showcar.com/>.
- [8] Rafael Lozano-Hemmer, *Perverting Technological Correctness*, Leonardo 29:1, 1996.
- [9] Markus Montola, *Games and Pervasive Games*, Page 7.
- [10] Masahiro Mori, 1970.
- [11] Jane McGonigal, "A Real Little Game: The Performance of Belief in Pervasive Play." *Digital Games Research Association (DiGRA) "Level Up" Conference Proceedings*. November 2003.
- [12] Marshall and Eric McLuhan, *Laws of Media: The New Science*, 1989
- [13] The Mirror (newspaper), *SatNav danger revealed: Navigation device blamed for causing 300,000 crashes*, Tanith Carey, 21/07/2008.
- [14] Pac-Manhattan, <http://pacmanhattan.com/about.php>, 2004.
- [15] Tychsen et al., *Live Action Role Playing Games*, page 255.

Computer Game Mods, Modders, Modding, and the Mod Scene

Walt Scacchi
Institute for Software Research
and
Center for Computer Games and Virtual Worlds
University of California, Irvine 92697-3455 USA
December 2009

Final version to appear in *First Monday*, <http://www.firstmonday.org>, Spring 2010.

Abstract

Computer games have increasingly been the focus of user-led innovations in the form of game mods. This paper examines how different kinds of socio-technical affordances serve to organize the actions of the people who develop and share their game mods. The affordances examined include customization and tailoring mechanisms, software and content copyright licenses, game software infrastructure and development tools, career contingencies and organizational practices of mod teams, and social worlds intersecting the mod scene. Numerous examples will be used to ground this review and highlight how such affordances can organize, facilitate or constrain what can be done. Overall, this study helps to provide a deeper understanding of how a web of associated affordances collectively serve to govern what mods get made, how modding practices emerge and flourish, and how modders and the game industry serve each others' interests, though not always in equivocal terms.

Introduction

Computer game mods are a leading form of user-led innovation in game design and game play experience. But modded games are not standalone systems, as they require the user to have an originally acquired or authorized copy of the unmodded game. Thus, there are questions of not only who creates what and who owns such modified games, but also whether or how the practice of game modding is controlled or governed by external parties to ultimately exploit the efforts of game modders. Conversely, if game mods are co-created by game studio developers and user-modders, then it may also be the case that modders create or enable developers, and thus modders govern some of the actions and choices of developers. In this article, we examine these notions by investigating an array of loosely coupled affordances among people, information resources, computing technologies, and institutional arrangements that are expressed or embodied through game mods, modding practices, modders, and the mod scene.

Affordances refer to situated, interactional properties between objects (tangible or symbolic) and actors that facilitate certain kinds of social interactions within a complex environment. The concept of affordances seeks to characterize aspects of complex work settings that facilitate how people interact though socio-technical systems like computer games. Computer-supported game play environments, when effective, afford new ways and means for collaborative learning [Gee 2008, Hayes and Games 2008, Scacchi, Nideffer and Adams 2008]. However, affordances are neither universal nor ubiquitous, nor are they experienced in the same way in different settings. Accordingly, an affordance that enables or encourages certain kinds of actions in one game mod setting, may inhibit or discourage similar actions in another setting. Modding practices observed for one game may differ from another game, so details and settings do matter. Subsequently, the focus in this article is directed at the interactions that facilitate collaborative activities between game modders who are geographically dispersed but share

access to online artifacts, networked information repositories and communication infrastructures, such as Web pages, Web sites, source code version servers, distributed file servers, and virtual private networks.

Affordances are enablers of collaboration and governance. For example, the online game modding artifacts, circumstances, and boundary objects can make different forms of governance obscure or coordination/control less visible [Star and Strauss 1999] and more remote. However, the information resources, social arrangements, technological system configurations, personal and professional interests that surround game mods, modding, and modders continually emerge as a web of associations or interrelated affordances that stabilize actions and practices [Latour 2003]. Consequently, our interest lies in examining where such associations stabilize and entrench the “social” surround of computer games in different settings, and at different levels of analysis [Kling and Scacchi 1982, Scacchi 2004, Scacchi 2008].

Governance can therefore be perceived through a web of affordances that either push it into a nearly invisible background, or into a visible confrontational foreground, in ways that may be historically situated or transcendent across space, time and circumstance. Thus looking at, into, or through a web of associations requires multiple lenses with different focal properties to discern how affordances widely available across the game community collectively act to govern who mods what, where, when, how and why.

Mods, Modders, Modding, and the Mod Scene

Depending on who you ask or where you look, game mods mean different things to different people. So if we seek to understand how game mods (objects), modding (activities or practices), modders (people working in particular situations), or the mod scene (the surrounding social world), we frame our study as one encompassing multiple affordances that naturally arise at different levels of granularity. Accordingly, we observe activities at levels that span (a) game mod embodiments, (b) game software and content licenses, (c) game modding software infrastructure and development tools, (d) career contingencies and organizational practices of game modders, and (e) social worlds intersecting the mod scene. While no claim is made that such levels are unique or definitive, their role is to help ground and reveal how governance of game mods, modding practices, modders, and mod scene occurs across a society of games.

We thus turn to examine modding at each of these levels, starting with what constitutes a mod.

Customizing, Tailoring, and Remixing Game Embodiments

Understanding what constitutes a game mod is not as simple as it might seem. Said differently, if you in some way alter a game, how its played, or on what it is played, have you created a game mod? Since we prefer not to presume some prior definition of game mods that might privilege the inclusion or discount the exclusion of some mods, modding practices, or modders, we choose to employ a wide-ranging view. So to start, we conceive of game mods as covering customizations, tailorings, and remixes of game embodiments, whether in the form of game content, software, or hardware denoting our space of interest.

As scholars like Dourish [2001] and Suchman [2007] have observed with other computing technologies, understanding game mods starts from observing how players interact with and reconfigure the game embodiments at their disposal. Computer games and game mods take on tangible form that reflect certain kinds of social computing experiences that are mediated by game system platforms [Montfort and Bogost 2009]. Modding is a “Do It Yourself” approach to technology

personalization that can establish both socio-technical and distributed cognitions for how to innovate by resting control over technology design from their producers. Modding is a form of *meta-gaming* — playing games for playing with the game systems. At least five types of game mods can be observed: user interface customization; game conversions; machinima and art mods; game computer customization; and game console hacking. Each enables different kinds of affordances that govern mods, modding practices, and modders.

User interfaces to games embody the practice and experience of interfacing users (game players) to the game system and play experience designed by game developers. Game developers act to constrain and govern what users can do, and what kinds of experience they can realize. Some users in turn seek to achieve some form of competitive advantage during game play by modding the user interface software for their game, when so enabled by game developers, to acquire or reveal additional information that the users believe will help their play performance and experience. User interface add-ons subsequently act as the medium through which game development studios can support mass customization of their game products, as a strategy for increasing the likelihood of product success through user satisfaction [Pine 1992]. Three kinds of user interface customizations can be observed. First and most common, is the player's ability to select, attire or accessorize the characters that embody a player's in-game identity. Second, is for players to customize the color palette and representational framing borders of the their game display within the human-computer interface, much like what can also be done with Web browsers and other end-user software applications.¹ Third, are user interface add-on components that modify the player's in-game information management dashboard that do not necessarily modify game play rules or functions. These add-ons provide additional information about game play that may enhance the game play experience, as well as increasing a player's sense of immersion or omniscience within the game world through sensory or perceptual expansion. This in turn enables awareness of game events not visible in the player's current in-game view [Taylor 2006], as the cognition required for sustained game play become more distributed [Taylor 2009]. Kow and Nardi [2010] describe their study of add-on modding practices for *World of Warcraft* in a cross-cultural setting, which helps reveal that cultural practices in different nations may encourage or not encourage such modding, as a reiteration of norms that tacitly govern social collectivity and conformity, and that eschew radical individuality and innovation.

Game Conversion Mods are perhaps the most common form of game mods.² Most such conversions are partial, in that they add or modify (a) in-game characters including user-controlled character appearance or capabilities, opponent bots, cheat bots, and non-player characters, (b) play objects like weapons, potions, spells, and other resources, (c) play levels,³ zones, terrains, or landscapes, (d) game rules,⁴ or (e) play mechanics.⁵ Some more ambitious modders go as far as to accomplish either (f) total conversions that create entirely new games from existing games of a kind that are not easily determined from the originating game, or (g) parodies that implicitly or explicitly spoof the content or play experience of one or more other games via reproduction and transformation. For example, one of the most widely distributed and played total game conversions is the *Counter-Strike* (CS) mod of the *Half-Life* first-person action game from Valve Software. As the success of the CS mod gave rise to millions of players preferring to play the mod over the original HL game, then other modders began to access the CS mod to further convert in part or full, to the point that Valve Software modified its game development and distribution business model to embrace game modding as part of the game play experience that is available to players who acquire a licensed copy of the HL product family.⁶ Valve has since marketed a number of CS variants that have sold over 10M copies as of 2008, thus denoting the most successful game conversion mod, as well as the most lucrative in terms of subsequent retail sales derived from a game mod.⁷ Other player-modders of *Half-Life* games have also benefited by access to meta-mods, such as *Garry's Mod* of *Half-Life 2*, which has evolved into a modding toolkit⁸ that has generated hundreds of game conversions and inventive game play mechanics. Though the success of a

single game conversion may not foretell a similar success of any subsequent mod (as no game conversion to date has reached anywhere the global success attained by CS), other game conversions represent innovations in game design and re-purposing. For example, the game *Chex Quest* is a conversion of the first-person shooter, *Doom*, into a “non-violent” game that was distributed in the Chex brand of cereal boxes on a CD targeted to young people and gamers.⁹ Another example is found in games converted to serve a purpose other than entertainment, such as the development and use of games for science, technology, and engineering applications. For instance, the *FabLab* game [Scacchi 2010] is a conversion of the *Unreal Tournament* game, from a first-person action shooter to a simulator for training semiconductor manufacturing technicians in diagnosing and treating potentially hazardous materials spills in a cleanroom environment. However, this conversion is not readily anticipated by knowledge of the *Unreal* games or underlying game engine. Finally, another class of game conversions are found in the form of game parodies — games created to reproduce and reframe the game play concepts and mechanics found in one or more popular games. For example, the game *WTF?!* is a 2D, Flash-based, side-scrolling, parody of the 3D *World of Warcraft* game that shares exportable character information from WoW.¹⁰ But *WTF?!* introduces new characters like Sigmund Freud, Karl Marx, and Mary Daly who serve to mod the WoW play experience through commentaries and quest directives that act to direct the player's attention respectively to construction of identity of player game characters, alternatives to capitalistic resource accumulation as game play goal, and dealing with the objectification of the female form within games.

Machinima and Art Mods can be viewed as the product of modding efforts that intend to mod play experience.¹¹ These products employ computer games as their creative media, such that these new media are mobilized for some other purpose (e.g., creating online cinema or interactive art exhibition). Machinima focuses attention to playing and replaying a game for the purpose of story telling, movie making,¹² or retelling of daunting or high efficiency game play experience [Lowood 2008]. Machinima is a form of modding the experience of playing a specific game through a recording of its visual play session history so as to achieve some other ends beyond the enjoyment (or frustration) of game play. These play session histories could then be modded via video editing or remixing with other media (e.g., audio recordings) to better enable cinematic storytelling or creative performance documentation. Art mods also modify the game play experience through manipulation, intervention, appropriation, or other creative transformation of a game's original visual content as it is consumed by users during during a play session. Artists working in the interactive medium of games have explored appropriation and intervention as tactics for using modded games as static, dynamic, or performance art works.¹³ Artists were among the first community of game players to embrace game hacks and patches: modifications to original game software functionality as a strategy for creating new works of art intended for curation and exhibition. The 1999 “Cracking the Maze” art exhibit curated by Anne-Marie Schleiner helped bring these modified games to light through an online exhibit that was structured as a C++ source code program for viewing or interaction in an Emacs-like text editor.¹⁴ Art mods can also incorporate alternative game control devices and user interface non-sequiturs.¹⁵

Custom gaming PCs are another expression of game system modding practices. Here players direct their efforts not to merely accessorizing their game platforms, but to go under the hood and to assemble, reassemble, or otherwise reconfigure their general-purpose personal computer into to one that is specialized for competitive game play or display in a multi-player environment [cf. Suchman 2007]. What results are “hot rod” and “custom car” PCs, computer hardware platforms modded for either maximum speed and performance, or special appearance for case mods [Simon 2007], rather than for mere operation of mainstream software applications or casual game play. Hot rod game PCs often feature high performance and over-clocked CPUs with liquid-cooling devices, custom-timed on-board memory chipsets, multiple GPU cards, and a variety of accelerator cards for in-game physics, networking, game audio, and so forth.¹⁶ These PCs embody a desire to achieve competitive game play

advantage for some, or sheer display of performance or signification of performance potential, much like automobiles are modded by their owners for the purpose of increasing their driving performance on the road, or as might be called for in a street racing situation.¹⁷ Though public practice of such situations with hot rod street cars is often isolated and criminally penalized when excessive, PC platform hot rodding is yet to achieve such distinction, public recognition, consequence, or cinematic embodiment. In contrast, PC case mods, which generally do not modify game functionality or performance, serves to signify a game player's interest or technological projection of self-identity onto their game play platform. Such projection denotes an unabashed choice to display one's enthusiasm, alignment, and commitment to game play as more than just entertainment, but as part of one's personal identity, fetish, cultural experience and life-style preference. Nonetheless, enthusiasts who do so mod their computers, also show off their machines in more public venues like regional LAN parties or game play conferences like QuakeCon.¹⁸ Custom gaming PCs thus represent a strategy for game play enthusiasts to proudly assert their embrace of game culture and the vendors that supply the means to display such embrace, though while being able to pose in stance that signifies independence from the governance telegraphed by corporately designed PCs.

Game console hacking is a practice whose purpose oftentimes seems to be in direct challenge to the authority of game console manufacturers that represent large, global corporate interests. Console hacking, in contrast to custom PC (re)configuration, is often focused not so much on how to improve competitive advantage in multi-player game play, but instead is focused on expanding the range of experiences that users may encounter through a game console [Consalvo 2007, Huang 2003]. For example, Huang's [2003] treatise on how to hack into a Microsoft Xbox console stresses how to unlock the potential of the Xbox as an alternative personal computer that could also support personal productivity applications or services, along with playing proprietary games acquired from Microsoft game vendors. However, Huang's study instructs readers in the practice of "reverse engineering" as a strategy to understand both how a computing platform was designed and how it operates in fine detail, as a basis for developing new innovative modifications or original platform designs, such as installing and running a Linux open source operating system (instead of Microsoft's proprietary closed source offering).¹⁹ While many game developers seek to protect their intellectual property from reverse engineering through end-user license agreements, whose terms attempt to prohibit such action under threat of legal action, reverse engineering is not legally prohibited nor discouraged by the Courts. Consequently, the practice of game console modding is often less focused on enabling players to achieve competitive advantage when playing retail computer games, but instead may encourage those few so inclined in the practice for how to understand and ultimately create computing innovations through reverse engineering²⁰ or other DIY hardware modifications.²¹ Console modding is thus an expression of game players who are willing to fore go the "protections" and quality assurances that console developers provide through product warranties, in order to experience the liberty, skill and knowledge acquisition, as well as potential to innovate, that mastery of reverse engineering affords. So while game console developers and retailers may seek to govern the actions that discourage users from looking inside of the gaming platforms, players who are willing to take responsibility for their actions (and not seek to defraud vendors due to false product failure warranty claims), can enjoy the freedom to learn how their gaming systems work in intimate detail and to potentially learn about hardware innovation with the support of others like-minded.

Software and Content Licenses

Players acquire computer games through retail outlets (online or offline stores). But what they pay for when they "buy" a game is a license to use the game product or product media. This license is commonly expressed as a copyright-based, end-user license agreement (EULA), much like it is for

most other software that consumers acquire. The terms and conditions of such license stipulate what rights and obligations a user of a licensed game copy realizes, as the game software and content is owned by the game's development studio, publisher, or distributor. So here we examine how a game's software and content license(s) can govern mods, modding, and modders.

First, for game conversion mods, it is common practice that the underlying game engine has one set of license terms and conditions to protect original work (e.g., no redistribution), while game mod can have a different set of terms and conditions as a derived work (e.g., redistribution allowed only for a game mod, but not for sale). In this regard, software licenses embody the business model that the game development studio or publisher seeks to embrace, rather than just a set of property rights and constraints. For example, in *Aion*, an MMOG from South Korean game studio NCSoft, no user created mods or add-ons are allowed, so attempting to incorporate such changes would conflict with its EULA and subsequently put such user-modders at risk of losing their access to networked *Aion* multi-player game play. In contrast, *WoW*, also an MMOG, allows for UI customization mods and add-ons only [Kow and Nardi 2010], but no other game conversions, no reverse engineering game engine, and no activity intended to bypass *WoW*'s encryption mechanisms.²² And, in one more variation, for games like *Unreal Tournament*, *Half-Life*, *NeverWinterNights*, *Civilization* and many others, the EULAs encourage modding and the free redistribution of mods without fee to others who must have a licensed game copy,²³ but no reverse engineering or redistribution of the game engine required to run the mods. This restriction in turns helps game companies realize the benefit of increased game sales by players who want to play with known mods, rather than with the unmodded game as sold at retail. Mods thus help improve games sales, revenue, and profits for the game development studio, publisher, and retailer (see note 7 for details).

Second, it is sometimes noted that game software licenses fundamentally differ based on whether they seek to insure copyright or copyleft, where the former provides closure and prevents certain freedoms to act, while the latter insures openness and certain freedoms — freedom of choice, freedom of expression, and freedom to modify and redistribute, plus obligation to propagate these freedoms. So it is sometimes stated that game conversion mods can be viewed and treated as free software. But such a position is an overstatement of fact in that free/open source software that is protected with a GPL license covers the source code and embedded documentation, but not game assets, contents, play mechanics, or rules of play, which can also be subject to protection through copyright, trademark, patent, trade secret, and restricted use. So as a growing number of games allow for modding and redistribution of mods, then we should expect to see a growth and diversification of the number of licenses, or EULA terms and conditions, that will constrain the rights and obligations that will apply to games, game engines, game assets, and game modders. Early signs of this have already begun to appear with heterogeneous licensed game development platforms like *Unity3D*, which itself is subject to more than a dozen different intellectual property (IP) licenses [Alspaugh, Asuncion, Scacchi 2009].

Similarly, much like some open source software, such as the Mozilla Firefox Web browser, open game engines may become subject to dual-mode, tri-mode, or multi-mode copyright licenses.²⁴ Finally, “open gaming licenses” have begun to appear which allow/restrict game configurations that include open source software along with proprietary data assets/artwork, game play rules, or game product identity.²⁵ In sum, game software and content licenses act to govern what mods can or cannot be made, by whom, and with what subsequent rights and obligations. But it is also clear that such governance is only effective where considered legitimate, as there are those who will ignore declared IP restrictions or act to crack open games to mod them in some way, and to share their results with others. Thus we should recognize that there is also resistance to governance, in the form of those who choose to ignore the license terms and conditions that come with closed, restricted games.

Game Software Infrastructure and Development Tools

Games are most often modded with tools that provide access to an internal, unencrypted representation of the game software²⁶ or game platform. While it might seem the case that game vendors would seek to discourage users from acquiring such tools, we observe a widespread contrary pattern. Game system developers are increasingly offering software tools for modifying the games they create or distribute, as a way to increase game sales and market share. SDKs and other modding tools provided to users by game development studios represent a contemporary business strategy for engaging users to help lead product innovation from outside the studio [von Hippel and Katz 2002, Jepperson 2005]. Once id Software and Epic Games, maker of the *Unreal* series of game, started to provide game play enthusiasts (who were also literate in computer programming) with software tools that would allow users to edit game content, play mechanics, rules, or other functionality, other competing game development studios were pressured to make similar offerings or face a possible competitive disadvantage in the marketplace. However, these tools do not provide access to the underlying source code that embodies the game engine—a large software program infrastructure that coordinates computer graphics, user interface controls, networking, game audio, access to middleware libraries for game physics, and so forth.

A tool support platform allows a game development studio to offer game products and services that users can mod. At the same time, studios seek to control access to the core IP that enables game play and modification. Accordingly, we can observe different strategies for how game development studios configure and support game modding platforms. For example, *World of Warcraft* from Blizzard Entertainment, provides a a UI Customization tool and a governance policy²⁷ and that enable WoW add-ons that reconfigure a WoW player's user interface dashboard, which in turn may provide an enhanced game play experience for technically accomplished players. But these add-ons do not modify or convert the WoW game into something entirely different, since Blizzard seeks to insure and control that users with or without add-ons have access to the same WoW in-game play setting and game play mechanics. In contrast, BioWare offers its *NeverWinterNights* game series with its Aurora Toolset and module construction toolkit,²⁸ FAQs, online tutorials, and in-game character profiles to help would be user-made content creators to modify NWN game play levels and characters. But creating new characters for such mods requires additional tools (e.g., modeling software like Autodesk 3D Studio) and corresponding specialist skills. Similarly, Epic Games has recently announced that it is distributing a suite of a dozen or so game development tools as the *Unreal Development Kit*,²⁹ which is said to be the same SDK that Epic previously marketed and distributed only to other game development studios that bought its commercial game software development licenses. However, the complexity and capabilities of such a tool suite mean that any one, or better said, any game development or modding team, can now access the tools to build commercial quality games, though mastering these tools appears to be a significant undertaking likely to be only of interest to highly committed, would-be game developers who are self-supported or self-organized.

Subsequently, we observe a range of corporate governance and control schemes that vary across game development firms, from (a) highly focused and limited at Blizzard for WoW, to (b) supported mediation in NWN by Bioware, to (c) do whatever you can if you have the skill, resources, team, and other resources to do so and sell your results, if you pay for such right with Epic Games. But all these schemes still stipulate control over the underlying game engine, such that it is out of reach to game players, as a way to protect key IP resources of the game studios.

In contrast to game modding platforms provided by game development studios, there are also alternatives provided by the end-user community. One approach can be seen with facilities provided in *Garry's Mod* mod-making package³⁰ that you can use to construct a variety of fanciful contraptions, to create comic books, to program game conversions, or to produce other kinds of user created content.

But this package requires that you own a licensed game like *Counter-Strike: Source*, *Half-Life 2* or *Day of Defeat: Source*, all from Valve Software. A different approach to end-user game development platforms can be found arising from free/open source software games and game engines. As noted above, the *Doom* and *Quake* games and game engines are released as free software that are subject to the GPL,³¹ once they were seen by id Software as having reached the end of their retail product cycle. Hundreds of games/engines have been developed and released for download starting from the free/open source software that was the platform of the original games. However, the content assets for many of these games (e.g., in-game artwork) are not covered by the GPL, and so user-developers must still acquire a licensed copy of the original game if its content is to be reused in some way. Nonetheless, some variants of the user-created GPL'd games now feature their own content that is limited/protected by Creative Commons licenses.

Career Contingencies and Organizational Practices of Modders Alone and Together

Who gets “paid” to make games when comparing developers versus modders? Developers who work for established game studios get paid a monetary salary and sometimes stock options (*financial capital*) to make game content, software, or play devices. Modders in general do not get a paid to make games, unless they are unusually successful by winning an open modding competition [Sotamaa 2007] or landing an angel/venture capital investment. However, modders can earn or acquire other kinds of career-oriented resources of professional or occupational value. These resources include deep game development knowledge, skills for how and when to apply such knowledge, status, community recognition and reputation, and other forms of *social capital* for the gifts of labor, creativity, expertise, and collaboration efforts they contribute [Portes 1998]. In this regard, both employed game developers and independent game modders can build on the efforts of one another in ways that are asymmetric and non-equivalent. Nonetheless, such an relationship is valuable to those who are committed to a career in game development inside or outside of the computer game industry. This is similar to what has been observed in a number of studies of open source software development projects, where in some cases, corporations that sponsor an open source project look to recruit and hire project contributors who make substantial and highly regarded software contributions to a project [Scacchi 2007].³² Furthermore, it has also been observed that developers who have a track record of success in open source software projects often get paid a higher monetary salary compared to their workmates who lack comparable experience [Hann, et al. 2002]. So it may be the case that game modders have the potential to invest in themselves as a way to acquire professional identity, accomplishments, and compensation if they choose to direct their career trajectory towards becoming a game developer in a studio. However this remains an open question for further study to substantiate, refine or refute. In the interim, what kind of investment is required of a game modder to enable such career contingencies?

To begin, why can't you get a job working for a game company if you are a software developer or artist? In simple terms, the computer game industry often looks to hire experienced game developers, rather than just game players or fans. Commercial game development is a risky venture, as most retail games often fail to recoup their development costs (as do many commercial software development efforts). Failure to produce a profitable game is all too familiar of an outcome, even for established game studios with skilled developers. Adding new staff with little or no prior game development experience implies the need for training and socialization, which is time consuming and may be unproductive. Game studios tend to hire developers who have some sort of record of game development. So without prior industry experience, starting as an independent game modder is the easiest way to determine whether one has the skill, disposition, and commitment to make games, even without a job to do so. Consequently, game modders must start and follow a self-motivated path.³³

The most direct way to become a game modder is through self-tutoring and self-organizing practices. Modding is a form of learning – learning how to mod, learning to be a game developer, learning to become a game content/software developer, learning computer game science outside or inside an academic setting, and more [El-Nasr and Smith 2006, Hayes and Games 2008, Scacchi 2004]. Modding is also a practice for learning how to work with others, especially on large, complex games/mods. Many would-be modders seek out others who have gone before them, much like game players at lower game play levels sometimes seek advice from players at higher levels for how best to proceed. In acting to become a modder or game developer, it may be reasonable to search out others like minded by joining a modding team, learning modding practices and mod team roles, and why you can't be in charge until you are recognized as being capable of leading.³⁴ Mod team efforts may also self organize around emergent leaders or “want to be” (W.T.B.) game component development leaders, as highlighted in the *Planeshift* open source MMOG development/modding project.³⁵

Modding is also a viable way to get a job in the game development industry. Epic Games for many years has encouraged would-be game developers to download its game editing tools (*UnrealEd*, the early precursor into what is now the *UDK* described earlier), start modding an *Unreal* game, and then perhaps get hired by a game development studio as a experienced game developer.³⁶ Similarly, to stand behind its advocacy of such a career pathway into a game development studio, Epic Games has also sponsored a series of mod-making contests, most recently under the name of “Make Something Unreal” that provides sizable cash prizes (with co-funding from Intel Corporation), or a full commercial license for *UDK*, so that a game conversion mod can be marketed and sold. Such an outcome essentially provides the winning mod team with venture seed capital, but without the venture capitalist's ownership of the new venture. Making game conversion mods may also open pathways for modders from a small game studio with limited resources to move upward into a large game studio with ample resources and multiple game development projects.³⁷ However, some scholars [Kücklich 2005, Somataa 2007] are skeptical about the value of modding, which they see as “playbour” (a coercion of leisurely play into work) or a commodification of leisure time, and subsequently conclude that modding work primarily benefits game companies by exploiting the productive efforts of game modders. However, if game modders are acting out of self-interest to establish a career path into jobs at game companies, and if they get to create games conversions that open new career opportunities in other industries, then it would seem that modders use modding as a way to exploit game development studio's desire to hire skilled game developers. Consequently, our view going forward may now need to reflect that modding as work can be productive to both independent mod makers and commercial game studios, but in different ways that may further reinforce and reproduce one another.

So our position is the marketplace of career contingencies for game developers and game modders governs through an invisible hand of individual, team, and corporate self-organizing interests to help select and guide who will be able to work for paid salary or who will accumulate skill, experience, and social capital as a game or game mod developer.

Social Worlds Intersecting the Mod Scene

Computer gaming is such a large-scale, globally diverse social endeavor that involves hundreds of millions of people as to merit its conception as a computer-centered social movement otherwise known as a *computerization movement* [Elliott and Kramer 2008, Scacchi 2008]. As a computerization movement, it can be employed as an analytical lens at the societal or social world level: a collection of transcendent activities, actors in different/multiple roles, and technical system configurations that are delimited by shared beliefs/ideologies, technical frames for explaining social action and system designs, and structural interests of organizations that contend for control over growth, resource accumulation, and sustainability. They can also be viewed in terms of what other social worlds

(communities or “scenes”) they intersect, which is the point of departure here. Let's consider how the computer game mod scene intersects with the Warez scene, and with the world of free/open source software development.³⁸

The Warez scene³⁹ focuses attention to activities that endeavor to modify protected software and other digital media (recorded music, movies, console circuit designs) so as to make them widely available for unfettered access and consumption. Games are one of the most commonly modified types of software that are transformed in the Warez scene into “pirated games” or game warez that are “illegally downloaded.” The Warez scene in a sense is focused on engaging a kind of meta-game that involves modding game software products. Game piracy has thus become recognized as a collective, decentralized and placeless endeavor (i.e., not a physical organization) that relies on torrent servers as its underground distribution venue for game warez. Game development studios, publishers, and some retailers all frequently lament their belief that game piracy is a kind of theft that is reducing their game sales and profits, as well as circumventing extant Digital Rights Management mechanisms [e.g., Greenburg and Irwin 2008]. As recent surveys of torrent-based downloads reveals, in 2008 the top 10 pirated games represented about 9M downloads, while in 2009 the top 5 pirated games represent more than 13M downloads, suggesting a substantial growth in interest in and access to such modded game products.⁴⁰ And these figures do not account for sharing or exchange of game warez on private peer-to-peer networks or darknets. Finally, it seems that in parts of the physical world like India and throughout Asia, game consoles can be purchased directly from retailers already modded with modchips that allow them to run game warez, including games not available in their national retail markets. This is not to impune some negative social connotation on other cultures, but instead to point out that not all national cultures share Western values or beliefs about what can be shared or what piracy means, and under what conditions or corporate institutions may be affected.

In the case of game warez and modded game consoles, it may be the situation that the developers and distributors of such products still benefit from pirated warez, if they can substantiate that such piracy is theft of products with economic value. Such theft can be treated as a legitimate business deduction when determining tax payments to national tax ministries. That is, piracy reduces the amount of profits that are subject to tax, since established losses due to pirated products (e.g., number of pirated games downloaded from torrents) may be tax deductible. Eliminating all game piracy might then actually work against the financial interests of game studios or publishers, if the people who download do not otherwise act to buy games that would now be unavailable to them. Consequently, downloading pirated games from torrents may just represent another channel for distributing products and realizing benefits, with certain costs.

The Warez scene appears increasingly effective at cracking closed games and distributing modded games (or if you prefer, pirated games) and game platforms compared to the capabilities of game studios, publishers, and console manufacturers. So is such accomplishment merely a sign of growing resistance on the part of game consumers, or does it represent an emerging transformation of the marketplace for how games and mods, modding practices, and modders are developed, distributed, played, remixed, reproduced and re-purposed?

Game mods, modding practices, and modders are in many ways quite similar to their counterparts in the world of free/open source software development. Modding is to games, like FOSS development is to software — they are increasingly becoming a part of mainstream technology development culture and practice. Modders are players of the games they construct, just like FOSS developers are also users of the systems they develop. There is no systematic distinction between developers and users in these communities, other than there are users/players that may contribute little beyond their usage, word of mouth they share with others, and their demand for more such systems. At FOSS portals like SourceForge.com, which in December 2009 indicates more than 380K projects are registered in its

repository, the domain of “games” appears as the third most popular project category with over 38K projects, after the domains of “software development” and “Internet.” Of these game-focused FOSS projects which develop either FOSS-based games, game engines, or game tools/SDKs, all of the top 50 have logged more than 1M downloads. So the intersection of games and FOSS covers a substantial social plane, as both modding and FOSS development are participatory, user-led modes of system development, and both rely on continual replenishment of new participants joining and migrating through project efforts, as well as new additions or modifications of content, functionality and end-user experience [Nieborg 2005, Scacchi 2004, Scacchi 2007, Scacchi 2008]. Mods and FOSS development projects are in many ways experiments to prototype alternative visions of what innovative systems might be in the near future,⁴¹ and so both are widely embraced and practiced primarily as a means for learning about new technologies, new system capabilities, new working relationships with potentially unfamiliar teammates from other cultures, and more [cf. Scacchi 2007].

Overall, game mods, game warez, and FOSS can be recognized as expressive forms of participatory culture. Though as we have found above, there are different modes for governing such participation depending on the affordances at hand. But in each situation, governance realizes an emerging transformation the marketplace of ideas and the means of production from centralized authority with corporate enterprises, to decentralized commons-based peer production [Benkler 2006]. Furthermore, sometimes such production may seek its own non-profit foundation or for-profit incorporation, or it may eschew any established organizational form in order to flourish in an alternative market that may not yet be recognized, or one that is being pursued in hope of preventing its move into a position of dominance.

Discussion

One way for viewing and understanding game mods is through the lens of ownership. Ownership influences or controls the allocation of game modding resources, markets or venues where game mods are transacted, or other decisions pertaining to which game mods are created, shared, played, and remixed/remodded by who, when, and where. Such a perspective centralizes governance with those who control ownership of property (games, game assets, engines, and SDKs) and the means of game mod production. Alternatively, a decentralized view of governance draws attention to intrinsic governance activities, patterns, and practices, where ownership is grounded in individual, small group, or project team action and resources. Based on the preceding examination of game mod embodiments, game software licenses, modding infrastructure and tools, career opportunities, and social worlds intersecting game modding, it seems that it is not the case that one of these two perspectives is more spot on compared to the other, as conditions and circumstances that situate game mods, modding, modders, and the mod scene are more heterogeneous than homogeneous, and sometimes less hegemonic than previous studies of game mods have implied. Instead, it may be better to examine the balances accomplished, remade, unmade, or actively prevented among a web of affordances that situate game mods, modding, modders, and mod scenes.

The affordances described and analyzed in this article all point to a web of associations that encourage, enact, exploit, or inhibit how mods, modding, modders, and mod scenes are governed. Governing this mod squad through the five types of affordances examined means that is much less likely that they are one-sided, predominantly in favor of either established game companies or independent self-organizing modders. Instead, what requires a better understanding is how modding as a cultural practice continually shifts how artifacts, embodiments, technical configurations, and arrangement of participants on all sides govern, value, and reproduce what can be accomplished through game mods.

Conclusions

In the study here, we started with the quest to examine how the mod squad constituted from game mods, modding practices, and modders are governed, and to what ends. Affordances of five different types serves as guideposts at different levels of analysis that helped to navigate the path forward. These affordances center about game mod embodiments, game licenses, game modding tools and infrastructure, career contingencies of game modders as would-be game developers, and social worlds that impinge on the game modding culture. Along the way, we examined, employed, or referenced the kinds of analytical constructs or results that others have previously employed in their studies of game mods.

We have seen that there are no single category of object or thing that uniquely denotes what a game mod is. Instead, mods can appear as game user interface add-ons, game conversions of various kinds and complexity, machinima and interactive art, custom gaming PCs, or hacked gaming consoles. Viewing and locating game mods within such a technological ecosystem provides a better perspective for understanding what mods are or can be, who makes them, how and why. Mods or mod embodiments are therefore actors that participate in governing the mod squad.

We have seen that game software licenses (and by implication, game platform and console hardware designs) embody the business models of game development studios. Further, game licenses are increasingly multi-modal and heterogeneous, thus challenging nearly all to determine who can mod what with what. This suggests that such licenses will become a fertile ground for debate regarding what kinds of mods can be made, what modding practices and tools will be provided or tolerated, and who can mod what. Such debates over licenses as a mode of governance and resistance may increasingly spill over into the Courts as well.

Game modding tools, SDKs, modchips, and impinging open source software alternatives for content creation or game engine transformation are collectively an emerging socio-technical media. And as new media, they serve stimulate, enact, or communicate continuing innovation, growth, and diversification by game development studios, game modders and others. The technologies of game modding thus help govern what kinds of mods and modders will emerge, flourish, or wither over time, as well as how the game industry and modding communities can cultivate and grow new markets for new game products.

Game modders are sometimes viewed as leisure laborers whose playful modding actions primarily contribute to the growth of economic rents collected by game studios. However, modders can also be viewed as independent and self-organizing actors who are acquiring the means for producing their own games or game conversions through self-serving investment in skill and knowledge acquisition, time, effort, and socialization with others like-minded. Modders can sometimes become entrepreneurial, and thus transform themselves from game consumers into producers. Consequently, the career contingencies of game modders cannot be simply assumed to be fully covered by some pre-determined set of producer-consumer arrangements or outcomes. Modders make mods through modding practices and technical means, and this in turn helps to produce more modders. Modding careers, contingencies, and pathways/trajectories thus help govern who becomes a modder and to what end.

The social world of game modding provides a system of beliefs, values, and norms that mod makers and users assimilate and reproduce in online modding forums and Web sites. This culture of game mods is enabled and embraced in the social arrangements and technical system configurations that frame what it means to be a game modder, to engage in game modding, and to make and play with a game mod. However, the world of game modding is not an island onto itself. Instead, like all social worlds, it is one that intersects some other worlds, and where and how these intersections occur is a locus of collective social movement, resource redistribution, market transformation and institutional conflict. Games are being modded by actors in the Warez scene in ways that are transforming who get

access to acquire and play what games where. This outcomes of this might be lamented as piracy or recognized as tax shelter on accumulated economic rents. Game modding is also influenced by, and influencing, the world of free/open source software development and its diverse software development projects and practices. Thus where, how, when, with whom, and with what the world of game modding intersects other social worlds that impinge on its boundaries will govern and frame why such intersections will arise, and to what end.

About the author

Walt Scacchi is a senior research scientist and research faculty member in the Institute for Software Research, and also Director of Research at the Center for Computer Games and Virtual Worlds, both at University of California, Irvine. He received a Ph.D. in Information and Computer Science at UC Irvine in 1981. From 1981-1998, he was a professor at the University of Southern California. Dr. Scacchi returned to UC Irvine in 1999. His research interests include open source software development, computer game culture and technology, virtual worlds for modeling and simulating complex engineering and business processes, developing decentralized heterogeneous information systems, software acquisition, and organizational analysis of system development projects. Dr. Scacchi is an active researcher with more than 150 research publications, and has directed more than 50 externally funded research projects. He also has had numerous consulting and visiting scientist positions with more than 25 firms or institutes, including four start-up ventures. His recent research activities and publications can be found at <http://www.ics.uci.edu/~wscacchi>.

Acknowledgements

Support for this research is supported through grants from the National Science Foundation #0534771 and #0808783, and also the Digital Industry Promotion Agency, Global Research and Development Collaboration Center, Daegu, South Korea. No endorsement implied.

Notes

- 1 For example, the Firefox Web browser provides “themes” and “personas” as a way to customize the appearance of a browser's user interface. See <http://support.mozilla.com/en-US/kb/Using+themes+with+Firefox?bl=n&s=themes>, accessed 30 March 2010.
- 2 Much, but not all, of the legacy of game conversion mods originates from the practices of the game software development studio, id Software, with legendary game developers John Carmack and John Romero. They tired of responding to game players requests for additional game features in their existing game products like *Wolfenstein 3-D* and *Doom*, while they sought to focus their attention to new games in development. Subsequently, they chose to hand over the source code to the *Doom* game to their end-user community, and then make the resulting source code subject to free software licensing rights, while still obligating user-modders to acquire the original retail version of the game to access in-game content assets. Many eager users embraced this open sharing of the *Doom* game engine source code, as it could reveal to a *Doom* player who was also a software developer, how the game engine was designed and coded, and thus how to modify or reproduce such an engine, or to build a new one, based on knowledge of the source code and skill in software development. The details of this story are more comprehensively described elsewhere, such as in *Masters of Doom* book by Kushner [2003], and also in studies by Au [2002], Kücklich [2005], and Morris [2003] among others.
- 3 For example, the game *Little Big Planet*, which features game level creation as one of its core game play mechanics, has realized the creation and upload/sharing of 1M levels within its first nine months of release, and now offers more than 1.3M user-rated levels. See http://www.littlebigplanet.com/en-us/game_guide/ps3/playing/community_levels, accessed 30 March 2010.
- 4 Nomic games (<http://www.nomic.net/>, accessed 30 March 2010) are those where changing the rules of a game is move, such that rule changing, generally with the agreement or vote of the game's players, becomes a play mechanic.
- 5 A popular Web portal where modders share and discuss their mods with modding fans who download, play, and rate the mods, can be found at <http://www.moddb.com>. These fans in turn may be just mod players, or they may be other modders who seek to learn about what others have accomplished, with whom to consult for advice, or with whom to try to join with in developing more ambitious game conversion mods. Mod fans or users also may consult <http://www.usercreated.org> for news and reviews of current mods, accessed 30 March 2010.
- 6 As with the previous note 2 on *Doom*, the history of CS is described elsewhere in many places, from Wikipedia <http://en.wikipedia.org/wiki/Counter-Strike>, to various game fan sites like <http://www.cstrike-planet.com/>, to game modding sites like Planet Counter-Strike at <http://planethalflife.gamespy.com/cs/> and others, accessed 30 March 2010.
- 7 See http://www.gamasutra.com/php-bin/news_index.php?story=21319 for disclosed sales figures from Valve Software, accessed 30 March 2010.
- 8 Game modding toolkits are also identified as “software development toolkits” (SDKs). However, they are not general purpose software development environments, but instead are specific to a game or family of games that share a common game engine. Software engineers designate such toolkits as domain-specific SDKs that center about the use of a domain-specific programming language, like *UnrealScript* for the *Unreal* game engine.
- 9 See http://chexquest.wikia.com/wiki/Chex_Quest and also <http://www.youtube.com/watch?v=C1A6dznZPM>, accessed 30 March 2010.
- 10 See <http://aoedipus.net/> for access to the *WTF?! game*, and also to <http://www.nideffer.net/promo/proj/wtf.html> for an overview. The !SDK also supports subsequent modding of *WTF?* -- see <http://nideffer.net/promo/proj/wtfe.html>, accessed 30 March 2010.
- 11 See <http://www.machinima.com> and <http://en.wikipedia.org/wiki/Machinima> for machinima, while for art mods see <http://www.gamescenes.org/>, http://www.selectparks.net/modules.php?name=News&new_topic=5, and http://en.wikipedia.org/wiki/Video_game_art for links to other descriptions and source examples, accessed 30 March 2010.
- 12 The game, *The Movies*, from Lionhead Studios, provides a game play experience that explicitly focuses on tools and techniques for movie-making, as well as enabling creation of parodies about movie-making studios and stars. Details at <http://www.lionhead.com/TheMovies/MakingMovies.aspx> (accessed 30 March 2010) encourage game players to make their own movies and post them on the Internet.

- 13 *Velvet-Strike* is an example of an interventionist art mod that alters the CS mod of *Half-Life* by artists Anne-Marie Schleiner, Joan Leandre and Brody Condon in 2002, so as to allow for inclusion of anti-war graffiti within the tactical small-group battleground of a CS game level/world. See <http://www.opensorcery.net/velvet-strike/>, accessed 30 March 2010.
- 14 See <http://switch.sjsu.edu/CrackingtheMaze/> and also <http://switch.sjsu.edu/web/v5n2/index2.html>, accessed 31 March 2010.
- 15 See art works by artists like Brody Condon <http://www.tmpspace.com/>, Robert Nideffer <http://www.nideffer.net>, and Eddo Stern <http://www.eddostern.com/>, as well as others found at net art portals like <http://www.selectparks.net>, accessed 30 March 2010.
- 16 Print-based and online periodicals like *MaximumPC* (<http://www.maximumpc.com/>, accessed 30 March 2010) and *CustomPC* (<http://www.custompc.co.uk/>, accessed 30 March 2010) focus attention and provide illustrated guidance for how to modify PCs to improve game run-time performance, and routinely employ performance intensive computer games like *Crysis* and *Fry Cry 2* (both of which can be modded) to demonstrate and document the results of PC systems before and after modding.
- 17 This kind of technological expression, identification, and eros is dramatized as popular culture in feature films like *The Fast and Furious* series (2001-2009), and earlier in *American Graffiti* (1973), *Two-Lane Blacktop* (1971), and *Hot Rod Girl* (1956).
- 18 See photos of examples from Quake2009 at <http://news.bigdownload.com/photos/quakecon-2009-pc-case-mods-pictures/2207346/> or from 2008 at <http://news.bigdownload.com/photos/quakecon-2008-pc-case-mod-gallery/955134/>, accessed 30 March 2010.
- 19 It should also be noted that in response to such dissemination of knowledge for how to hack into the Xbox system, that Microsoft invested a great deal of effort and resources to insure its subsequent Xbox 360 game platform could not be so readily hacked, and it seems to have mostly succeeded in these efforts, though to the disadvantage of its users, who must consequently look elsewhere to learn about reverse engineering and platform hacking as a basis for system innovation or customization. In this regard, Microsoft's actions may suggest it does not trust the consumers of its products to engage in activities that might improve their personal situation, but instead acts to protect what Microsoft perceives to be its corporate interests within its dispersed intellectual property.
- 20 Such practice is to be distinguished from suspect or illegal misappropriation of trade secrets through covert acquisition of materials or documents from insiders or other illicit sources, rather than as a consumer who is inquisitive and willing to void their product warranty to look inside the back box to see what's there, and who is willing to exercise the freedom to figure out how it does or doesn't operate, and who can create or modify what is found to achieve new operational capabilities.
- 21 Console hardware may also be modded through the incorporation of custom integrated circuits ("modchips") which circumvent vendor-imposed limitations or copy protection mechanisms. For example, see <http://www.modchip.com>, <http://www.mod-chip.com>, <http://www.modchipstore.com>, or <http://en.wikipedia.org/wiki/Modchip> among others, accessed 30 March 2010.
- 22 Prior to the release of WoW, Blizzard Entertainment actively pursued and ultimately shut down the independent development of a Battel.Net server portal called bnetd.org that was claimed to have been reverse engineered from public information. But Blizzard successfully prosecuted that such modding entailed work-around or breakage of in-game anticircumvention mechanism, and that such infringement made the bnetd.org modders legally culpable. The Court then ruled in favor of Blizzard Entertainment and bnetd.org was shut down. See Davidson & Assoc. et al., v. Jung et al., 422 F.3d 630, 633 (8th. Cir. 2005).
- 23 Kücklich [2005] reports the EULA for the Half-Life SDK "...hereby grants Licensee a nonexclusive, royalty-free, terminable, worldwide, non-transferable license to: (a) use, reproduce and modify the SDK in source code form, solely to develop a Mod; and (b) reproduce, distribute and license the Mod in object code form, solely to licensed end users of Half-Life, without charge."

- 24 Information on the Mozilla tri-license for the Firefox source code (the MPL as a minimally restricted license, the GPLv2 as a reciprocal license, and the LGPLv2.1 as a library-style license) can be found at <http://www.mozilla.org/MPL/>, accessed 30 March 2010.
- 25 See the Open Gaming Foundation at <http://www.opengamingfoundation.org/licenses.html> and also the d20 Game System license from the Wizards of the Coast at <http://www.wizards.com/default.asp?x=d20/welcome>, accessed 30 March 2010.
- 26 In a game like *Doom*, this internal representation is denoted by a WAD file [Kushner 2003, Morris 2003].
- 27 See <http://www.worldofwarcraft.com/ui/> and <http://www.worldofwarcraft.com/policy/ui.html>, accessed 30 March 2010.
- 28 See <http://nwn.bioware.com/builders/>, accessed 30 March 2010.
- 29 See <http://www.udk.com/>, accessed 30 March 2010.
- 30 See <http://www.garrysmud.com/>, accessed 30 March 2010.
- 31 See <http://www.idsoftware.com/business/techdownloads/>, accessed 30 March 2010.
- 32 The game development studio, Crytek, has begun to offer job interviews at their company as a prize for winning a game modding contest that they host. Further information at <http://crymod.com/thread.php?threadid=56783>, accessed 30 March 2010.
- 33 For example, the lead designer of the *Civilization V* game from Firaxis Studios, Jon Shaffer, started his career as a modder of earlier *Civilization* games. His mods were recognized as being of sufficient quality, and his academic studies in European, 19th Century German, and World War II history aligned with his *Civilization* modding interests. Subsequently, he was recruited to work at Firaxis, and has become a lead designer there. See [Murdoch and Wilson 2010] for details.
- 34 Though now moved into archived status, the description of The Layman's Guide to Mod Making at http://wiki.beyondunreal.com/Legacy:Making_Mods details what's involved in team-based mod making efforts. Similarly, Postigo's [2007] study of the Home Front Mod team for *BattleField 1942* identifies more than 25 team members occupying 18 self-selected roles in mod making.
- 35 See <http://www.planeshift.it/recruitment.html> (accessed 30 March 2010) or [Scacchi 2004] for details.
- 36 See Scacchi [2004], p.64. Also see where such encouragement in modding as a career development strategy is reiterated at <http://www.makesomethingunreal.com/overview.aspx>, where it is stated, "...Nearly half of the people who work on Epic Games' development team were former mod-makers! We want to help you make it to the big leagues as well. Think you have what it takes? Check out the [Epic Jobs](#) page." accessed 30 March 2010.
- 37 The development of the *Chex Quest* game conversion by a small game studio did allow for one of its first-time mod maker's, Chuck Jacobi, to subsequently become lead game artist for Electronic Arts LA, a major game studio. See [Lathi 2009] for details.
- 38 Scacchi [2008] also draws comparison of the world of computer games, FOSS, and scientific research computing.
- 39 For an introduction, history, and sample practices of the Warez scene, see http://en.wikipedia.org/wiki/Warez_scene, <http://www.warezscene.org/>, <http://thewarezscene.org/forums/>, or <http://torrentfreak.com/interview-with-a-warez-scene-releaser/>, accessed 30 March 2010.
- 40 See <http://torrentfreak.com/top-10-most-pirated-games-of-2008-081204/> for 2008, and for 2009 see <http://torrentfreak.com/the-most-pirated-games-of-2009-091227/>, accessed 30 March 2010.
- 41 As another major area of FOSS development activity (at SourceForge and elsewhere) is in the domain of Science/Engineering, we should not be surprised to soon see new FOSS game engines or game physics libraries being modded to support more complex physical computations beyond forces and motions, and eventually include electro-

magnetism, quantum chromodynamics, and celestial mechanics, as well as chemical, biological, geological, and meteorological processes. Conversely, as such capabilities come into being, then we expect scientists and students working in such areas to increasingly see their experimental studies employing these new science game mods [Mayo 2007].

References

Thomas Alspaugh, Hazel Asuncion, and Walt Scacchi, 2009, "Intellectual Property Rights Requirements for Heterogeneously Licensed Systems," in *Proc. 17th. Intern. Conf. Requirements Engineering (RE09)*, Atlanta, GA, ACM Press, 24-33, September 2009.

Yochai Benkler, 2006. *The Wealth of Networks: How Social Production Transforms Markets and Freedom*, Yale University Press, New Haven, CT.

Mia Consalvo, 2007. *Cheating: Gaining Advantage in Videogames*, MIT Press, Cambridge, MA.

Rob Crossley, 2008. "The Most Pirated Games in 2008," *EDGE*, 8 December 2008. <http://www.next-gen.biz/news/the-most-pirated-games-2008>, accessed 23 December 2009.

Paul Dourish, 2001. *Where the Action Is: The Foundations of Embodied Interaction*, MIT Press, Cambridge, MA.

Margaret S. Elliott and Kenneth L. Kraemer (Eds.), 2008. *Computerization Movements and Technology Diffusion: From Mainframes to Ubiquitous Computing*, ASIST Monograph Series, Information Today Inc., Medford, New Jersey.

Magy Seif El-Nasr and Brian K. Smith, "Learning Through Game Modding", *ACM Computers in Entertainment*, 4(1). Article 3B.

James Paul Gee, 2008. "Video Games and Embodiment," *Games and Culture*, Jul 2008; vol. 3: pp. 253–263.

Andy Greenburg and Mary Jane Irwin, 2008. "Spore's Piracy Problem," *Forbes.com* http://www.forbes.com/2008/09/12/spore-drm-piracy-tech-security-cx_ag_mji_0912spore.html?partner=alerts, accessed 23 December 2009.

Elizabeth Hayes and I.A. Games, 2008. "Making Computer Games and Design Thinking: A Review of Current Software and Strategies," *Games and Culture* Jul 2008; vol. 3: pp. 309 - 332.

Il-Horn Hann, Jeffrey Roberts, Sandra Slaughter, and Roy Fielding, 2002. "Economic Incentives for Participating in Open Source Software Projects," in *Proc. Twenty-Third Intern. Conf. Information Systems*, 365-372.

Andrew “bunnie” Huang, 2003. *Hacking the Xbox: An Introduction to Reverse Engineering*, No Starch Press, San Francisco, CA.

Lars Bo Jepperson, 2005, “User Toolkits for Innovation: Consumers Support Each Other,” *Journal of Product Innovation Management*. 22, 347–362.

Rob Kling and Walt Scacchi, 1982, “The Web of Computing: Computer Technology as Social Organization”, in M. Yovits, (Ed.), *Advances in Computers*, Vol 21, 3-75.

Yong Ming Kow and Bonnie Nardi, 2010. “Culture and Creativity: *World of Warcraft* Modding in China and the US,” in W.S. Bainbridge (Ed.), *Online Worlds: Convergence of the Real and the Virtual*, Springer, New York. 21-42, (to appear).

Julian Kücklich, 2005. “Precarious playbour: Modders and the digital games industry,” *Fibreculture*, number 5, at <http://journal.fibreculture.org/issue5/kucklich.html>, accessed 22 December 2009.

David Kushner, 2003. *Masters of Doom: How Two Guys Created an Empire and Transformed Pop Culture*, Random House, New York.

Evan Lathi, 2009. Chex Quest: Never Forget, *PC Gamer*, 190(16), August, p. 96.

Bruno Latour, 2005, *Reassembling the Social: An Introduction to Actor-Network Theory*, Oxford University Press, New York.

Henry Lowood, 2008. “Found Technology: Players as Innovators in the Making of Machinima.” in Tara McPherson (Ed.), *Digital Youth, Innovation, and the Unexpected*. The John D. and Catherine T. MacArthur Foundation Series on Digital Media and Learning. MIT Press, Cambridge, MA, 165–196.

Paul Marino, 2004. *3D Game-Based Filmmaking: The Art of Machinima*. Paraglyph Press, Scottsdale, AZ.

Merrilea J. Mayo, 2007. “Games for Science and Engineering Education,” *Communications of the ACM*, 50(7), 31-35, July.

Nick Montfort and Ian Bogost, 2009. *Racing the Beam: The Atari Video Computer System*, MIT Press, Cambridge, MA.

Sue Morris, 2003. “WADs, Bots, and Mods: Multiplayer FPS Games and Co-creative Media,” *Level Up Conference Proceedings: 2003 Digital Games Research Association Conference*, Utrecht, University of Utrecht, November.

Julian Murdoch and Jason Wilson, 2010. “The Education of Civilization V,” *GamePro*, 259, 59-69, April.

David Nieborg, 2005. “Am I mod or not? — An analysis of first person shooter modification culture,” paper presented at *Creative Gamers Seminar — Exploring Participatory Culture in Gaming*,

University of Tampere, Finland (14–15 January).

Joseph Pine, 1992. *Mass Customization: The New Frontier in Business Competition*. Harvard Business School Press, Boston, MA.

Alejandro Portes, 1998. "Social Capital: Its Origins and Application in Modern Sociology," *Annual Review of Sociology*, 24, 1-24.

Hector Postigo, 2007. "Of Mods and Modders: Chasing Down the Value of Fan-Based Digital Game Modifications," *Games and Culture*, Oct 2007; vol. 2: pp. 300 – 313.

Hector Postigo, 2008. Video game appropriation through modifications: Attitudes concerning intellectual property among modders and fans. *Convergence*, 14, 59–74.

Walt Scacchi, 2004. "[Free/Open Source Software Development Practices in the Computer Game Community](#)," *IEEE Software*, 21(1), 59-67, January/February 2004.

Walt Scacchi, 2007. "Free/Open Source Software Development: Recent Research Results and Emerging Opportunities," *Proc. European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, 459-468, September 2007.

Walt Scacchi, 2008. "Emerging Patterns of Intersection and Segmentation when Computerization Movements Interact," in M.S. Elliott and K.L. Kraemer (Eds.), *Computerization Movements and Technology Diffusion: From Mainframes to Ubiquitous Computing*, ASIST Monograph Series, Information Today, Inc., Medford, New Jersey, 381-404.

Walt Scacchi, 2010, "Game-Based Virtual Worlds as Decentralized Virtual Activity Systems," in W.S. Bainbridge (Ed.), *Online Worlds: Convergence of the Real and the Virtual*, Springer, New York. 223-234, (to appear).

Walt Scacchi, Robert Nideffer, and Joe Adams, 2008. "A Collaborative Science Learning Game Environment for Informal Science Education: *DinoQuest Online*," in P. Ciancarini, R. Nakatsu, M. Rauterberg, M. Roccetti (Eds.), *New Frontiers for Entertainment Computing*, IFIP International Federation for Information Processing Series, Volume 279; Boston: Springer, 71–82.

Bart Simon 2007. "Geek Chic: Machine Aesthetics, Digital Gaming, and the Cultural Politics of the Case Mod", *Games and Culture*, Jul 2007; vol. 2, 175-193.

Olli Sotamaa, 2007. "On Modder Labour, Commodification of Play, and Mod Competitions," *First Monday*, 12(9).

Lucy Suchman, 2007. *Human-Machine Reconfigurations: Plans and Situated Actions (2nd Edition)*, Cambridge University Press, New York.

Susan Leigh Star and Amselm Strauss, 1999. "Layers of Silence, Arenas of Voice: The Ecology

of Visible and Invisible Work”, *Computer-Supported Cooperative Work*, 8(1/2), 9-30, March 1999.

T. L. Taylor, 2006. “Does WoW Change Everything?: How a PvP Server, Multinational Player Base, and Surveillance Mod Scene Caused Me Pause”, *Games and Culture*, Oct 2006; vol. 1: pp. 318-337.

T.L. Taylor, 2009. “The Assemblage of Play,” *Games and Culture*, 4(1), 331-339, October.

Eric von Hippel and Ralph Katz, 2002. “Shifting innovation to users via toolkits,” *Management Science*, 48(7), 821-833.

Software Licenses, Open Source Components, and Open Architectures

Thomas A. Alspaugh, Hazeline U. Asuncion, and Walt Scacchi

Institute for Software Research

University of California, Irvine

Irvine, CA 92697-3455 USA

{alspaugh,hasuncion,wcacchi}@ics.uci.edu

Abstract

A substantial number of enterprises and independent software vendors are adopting a strategy in which software-intensive systems are developed with an open architecture (OA) that may contain open source software (OSS) components or components with open APIs. The emerging challenge is to realize the benefits of openness when components are subject to different copyright or property licenses. In this paper we identify key properties of OSS licenses, present a license analysis scheme to identify license conflicts arising from composed software elements, and apply it to provide guidance for software architectural design choices whose goal is to enable specific licensed component configurations. Our scheme has been implemented in an operational environment and demonstrates a practical, automated solution to the problem of determining overall rights and obligations for alternative OAs.

1. Introduction

It has been common for OSS projects to require that developers contribute their work under conditions that ensure the project can license its products under a specific OSS license. For example, the Apache Contributor License Agreement grants enough rights to the Apache Software Foundation for the foundation to license the resulting systems under the Apache License. This sort of license configuration, in which the rights to a system's components are homogenously granted and the system has a well-defined OSS license, was the norm and continues to this day.

However, we more and more commonly see a different license configuration, in which the components of a system do not have the same license. The resulting system may not have any recognized OSS license at all—in fact, our research indicates this is the most likely

outcome—but instead, if all goes well in its design, there will be enough rights available in the system so that it can be used and distributed, and perhaps modified by others and sublicensed, if the corresponding obligations are met. These obligations are likely to differ for components with different licenses; a BSD (Berkeley Software Distribution) licensed component must preserve its copyright notices when made part of the system, for example, while the source code for a modified component covered by MPL (the Mozilla Public License) must be made public, and a component with a reciprocal license such as the Free Software Foundation's GPL (General Public License) might carry the obligation to distribute the source code of that component but also of other components that constitute “a whole which is a work based on” the GPL'd component. The obligations may conflict, as when a GPL'd component's reciprocal obligation to publish source code of other components is combined with a proprietary license's prohibition of publishing source code, in which case there may be no rights available for the system as a whole, not even the right of use, because the obligations of the licenses that would permit use of its components cannot simultaneously be met.

The central problem we examine and explain in this paper is to identify principles of software architecture and software licenses that facilitate or inhibit success of the OA strategy when OSS and other software components with open APIs are employed. This is the knowledge we seek to develop and deliver. Without such knowledge, it is unlikely that an OA that is clean, robust, transparent, and extensible can be readily produced. On a broader scale, this paper seeks to explore and answer the following kinds of research questions:

- What license applies to an OA system composed with components with different licenses?
- How do alternative OSS licenses facilitate or inhibit the development of OA systems?

- How should software license constraints be specified so it is possible to automatically determine the overall set of rights and obligations associated with a configured software system architecture?

This paper may help establish a foundation for how to analyze and evaluate dependencies that might arise when seeking to develop software systems that embody an OA when different types of software components or software licenses are being considered for integration into an overall system configuration.

In the remainder of this paper, we examine software licensing constraints. This is followed by an analysis of how these constraints can interact in order to determine the overall license constraints applicable to the configured system architecture. Next, we describe an operational environment that demonstrates automatic determination of license constraints associated with a configured system architecture, and thus offers a solution to the problem we face. We close with a discussion of the conclusions that follow.

2. Background

There is little explicit guidance or reliance on systematic empirical studies for how best to develop, deploy, and sustain complex software systems when different OA and OSS objectives are at hand. Instead, we find narratives that provide ample motivation and belief in the promise and potential of OA and OSS without consideration of what challenges may lie ahead in realizing OA and OSS strategies. Ven [2008] is a recent exception.

We believe that a primary challenge to be addressed is how to determine whether a system, composed of subsystems and components each with specific OSS or proprietary licenses, and integrated in the system's planned configuration, is or is not open, and what license constraints apply to the configured system as a whole. This challenge comprises not only evaluating an existing system at run-time, but also at design-time and build-time for a proposed system to ensure that the result is "open" under the desired definition, and that only the acceptable licenses apply; and also understanding which licenses are acceptable in this context. Because there are a range of types and variants of licenses [cf. OSI 2008], each of which may affect a system in different ways, and because there are a number of different kinds of OSS-related components and ways of combining them that affect the licensing issue, a first necessary step is to understand the kinds of software elements that constitute a software architecture, and what kinds of licenses may encumber these elements or their overall configuration.

OA seems to simply mean software system architectures incorporating OSS components and open application program interfaces (APIs). But not all software system architectures incorporating OSS components and open APIs will produce an OA, since the openness of an OA depends on: (a) how/why OSS and open APIs are located within the system architecture, (b) how OSS and open APIs are implemented, embedded, or interconnected, (c) whether the copyright (Intellectual Property) licenses assigned to different OSS components encumber all/part of a software system's architecture into which they are integrated, and (d) the fact that many alternative architectural configurations and APIs exist that may or may not produce an OA [cf. Antón and Alspaugh 2007, Scacchi and Alspaugh 2008]. Subsequently, we believe this can lead to situations in which new software development or acquisition requirements stipulate a software system with an OA and OSS, but the resulting software system may or may not embody an OA. This can occur when the architectural design of a system constrains system requirements—raising the question of what requirements can be satisfied by a given system architecture, when requirements stipulate specific types or instances of OSS (e.g., Web browsers, content management servers) to be employed, or what architecture style [Bass, Clements, and Kazman 2003] is implied by a given set of system requirements.

Thus, given the goal of realizing an OA and OSS strategy together with the use of OSS components and open APIs, it is unclear how to best align acquisition, system requirements, software architectures, and OSS elements across different software license regimes to achieve this goal [Scacchi and Alspaugh 2008].

3. Understanding open architectures

The statement that a system is intended to embody an open architecture using open software technologies like OSS and APIs, does not clearly indicate what possible mix of software elements may be configured into such a system. To help explain this, we first identify what kinds of software elements are included in common software architectures whether they are open or closed [cf. Bass, Clements, Kazman 2003].

1. Software source code components – (a) standalone programs, (b) libraries, frameworks, or middleware, (c) inter-application script code (e.g., C shell scripts) and (d) intra-application script code (e.g., to create Rich Internet Applications using domain-specific languages (e.g., XUL for Firefox Web

browser [Feldt 2007] or “mashups” [Nelson and Churchill 2006]).

2. Executable components -- These are programs for which the software is in binary form, and its source code may not be open for access, review, modification, and possible redistribution. Executable binaries can be viewed as “derived works” [Rosen 2005].
3. Application program interfaces/APIs – The availability of externally visible and accessible APIs to which independently developed components can be connected is the minimum condition required to form an “open system” [Meyers and Obendorf 2001].
4. Software connectors – In addition to APIs, these may be software either from libraries, frameworks, or application script code whose intended purpose is to provide a standard or reusable way of associating programs, data repositories, or remote services through common interfaces. The High Level Architecture (HLA) is an example of a software connector scheme [Kuhl, Weatherly, Damann 2000], as are CORBA, Microsoft’s .NET, Enterprise Java Beans, and LGPL libraries.
5. Configured system or sub-system architectures – These are software systems that can be built to conform to an explicit architectural design. They include software source code components, executable components, APIs, and connectors that are organized in a way that may conform to a known “architectural style” such as the Representational State Transfer [Fielding and Taylor 2002] for Web-based client-server applications, or may represent an original or ad hoc architectural pattern [Bass 2003]. Each of the software elements, and the pattern in which they are arranged and inter-linked, can all be specified, analyzed, and documented using an Architecture Description Language and ADL-based support tools [Bass 2003, Medvidovic 1999].

Figure 1 provides an overall view of an archetypal software architecture for a configured system that includes and identifies each of the software elements above, as well as including free/open source software (e.g., Gnome Evolution) and closed source software (WordPerfect) components. In simple terms, the configured system consists of software components (grey boxes in the Figure) that include a Mozilla Web browser, Gnome Evolution email client, and WordPerfect word processor, all running on a Linux operating system that can access file, print, and other remote networked servers (e.g. an Apache Web server). These

components are interrelated through a set of software connectors (ellipses in the Figure) that connect the interfaces of software components (small white boxes attached to a component) and link them together. Modern day enterprise systems or command and control systems will generally have more complex architectures and a more diverse mix of software components than shown in the figure here. As we examine next, even this simple architecture raises a number of OSS licensing issues that constrain the extent of openness that may be realized in a configured OA.

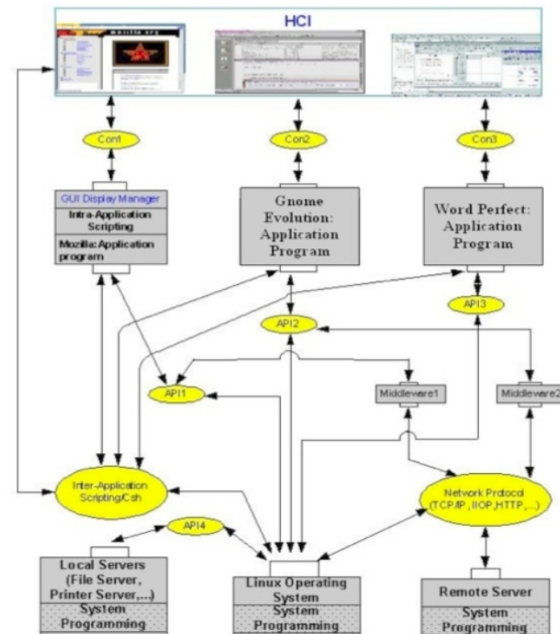


Figure 1. An archetypal software architecture depicting components (grey boxes), connectors (ellipses), interfaces (small boxes on components), and data/control links

4. Understanding open software licenses

A particularly knotty challenge is the problem of licenses in OSS and OA. There are a number of different OSS licenses, and their number continues to grow. Each license stipulates different constraints attached to software components that bear it. External references are available which describe and explain many different licenses that are now in use with OSS [Fontana 2008, OSI 2008, Rosen 2005, St. Laurent 2004].

More and more software systems are designed, built, released, and distributed as OAs composed of components from different sources, some proprietary and others not. Systems include components that are statically bound or interconnected at build-time, while other

components may only be dynamically linked for execution at run-time, and thus might not be included as part of a software release or distribution. Software components in such systems evolve not only by ongoing maintenance, but also by architectural refactoring, alternative component interconnections, and component replacement (via maintenance patches, installation of new versions, or migration to new technologies). Software components in such systems may be subject to different software licenses, and later versions of a component may be subject to different licenses (e.g., from CDDL (Sun's Common Development and Distribution License) to GPL, or from GPLv2 to GPLv3).

Software systems with open architectures are subject to different software licenses than may be common with traditional proprietary, closed source systems from a single vendor. Software architects/developers must increasingly attend to how they design, develop, and deploy software systems that may be subject to multiple, possibly conflicting software licenses. We see architects, developers, software acquisition managers, and others concerned with OAs as falling into three groups. The first group pays little or no heed to license conflicts and obligations; they simply focus on the other goals of the system. Those in the second group have assets and resources, and to protect these they may have an army of lawyers to advise them on license issues and other potential vulnerabilities; or they may constrain the design of their systems so that only a small number of software licenses (possibly just one) are involved, excluding components with other licenses independent of whether such components represent a more effective or more efficient solution. The third group falls between these two extremes; members of this group want to design, develop, and distribute the best systems possible, while respecting the constraints associated with different software component licenses. Their goal is a configured OA system that meets all its goals, and for which all the license obligations for the needed copyright rights are satisfied. It is this third group that needs the guidance the present work seeks to provide.

There has been an explosion in the number, type, and variants of software licenses, especially with open source software (cf. OSI 2008). Software components are now available subject to licenses such as the General Public License (GPL), Mozilla Public License (MPL), Apache Public License, (APL), Academic licenses (e.g., BSD, MIT), Creative Commons, Artistic, and others as well as Public Domain (either via explicit declaration or by expiration of prior copyright license). Furthermore, licenses such as these can evolve, resulting in new license versions over time. But no matter

their diversity, software licenses represent a legally enforceable contract that is recognized by government agencies, corporate enterprises, individuals, and judicial courts, and thus they cannot be taken trivially. As a consequence, software licenses constrain open architectures, and thus architectural design decisions.

So how might we support the diverse needs of different software developers, with respect to their need to design, develop, and deploy configured software systems with different, possibly conflicting licenses for the software components they employ? Is it possible to provide automated means for helping software developers determine what constraints will result at design-time, build-time, or run-time when their configured system architectures employ diverse licensed components? These are the kind of questions we address in this paper.

4.1. Software licenses: Rights and obligations

Copyright, the common basis for software licenses, gives the original author of a work certain exclusive rights, which for software include the right to use, copy, modify, merge, publication, distribution, sub-licensing, and sell copies. These rights may be licensed to others; the rights may be licensed individually or in groups, and either exclusively so that no one else can exercise them or (more commonly) non-exclusively. After a period of years, the rights enter the public domain, but until then the only way for anyone other than the author to have any of the copyright rights is to license them.

Licenses may impose obligations that must be met in order for the licensee to realize the assigned rights. Commonly cited obligations include the obligation to buy a legal copy to use and not distribute copies (proprietary licenses); the obligation to preserve copyright and license notices (academic licenses); the obligation to publish at no cost source code you modify (MPL); or the reciprocal obligation to publish all source code included at build-time or statically linked (GPL).

Licenses may provide for the creation of derivative works (e.g., a transformation or adaptation of existing software) or collective works (e.g., a Linux distribution that combines software from many independent sources) from the original work, by granting those rights possibly with corresponding obligations.

In addition, the author of an original work can make it available under more than one license, enabling the work's distribution to different audiences with different needs. For example, one licensee might be happy to pay a license fee in order to be able to distribute the work as part of a proprietary product whose source

code is not published, while another might need to license the work under MPL rather than GPL in order to have consistent licensing across a system. Thus we now see software distributed under any one of several licenses, with the licensee choosing from two (“dual license”) or three (Mozilla’s “tri-license”) licenses.

The basic relationship between software license rights and obligations can be summarized as follows: if you meet the specified obligations, then you get the specified rights. So, informally, for the academic licenses, if you retain the copyright notice, list of license conditions, and disclaimer, then you can use, modify, merge, sub-license, etc. For MPL, if you publish modified source code and sub-licensed derived works under MPL, then you get all the MPL rights. And so forth for other licenses. However, one thing we have learned from our efforts to carefully analyze and lay out the obligations and rights pertaining to each license is that license details are difficult to comprehend and track—it is easy to get confused or make mistakes. Some of the OSS licenses were written by developers, and often these turn out to be incomplete and legally ambiguous; others, usually more recent, were written by lawyers, and are more exact and complete but can be difficult for non-lawyers to grasp. The challenge is multiplied when dealing with configured system architectures that compose multiple components with heterogeneous licenses, so that the need for legal interpretations begins to seem inevitable [cf. Fontana 2008, Rosen 2005]. Therefore, one of our goals is to make it possible to architect software systems of heterogeneously-licensed components without necessarily consulting legal counsel. Similarly, such a goal is best realized with automated support that can help architects understand design choices across components with different licenses, and that can provide support for testing build-time releases and run-time distributions to make sure they achieve the specified rights by satisfying the corresponding obligations.

4.2. Expressing software licenses

Historically, most software systems, including OSS systems, were entirely under a single software license. However, we now see more and more software systems being proposed, built, or distributed with components that are under various licenses. Such systems may no longer be covered by a single license, unless such a licensing constraint is stipulated at design-time, and enforced at build-time and run-time. But when components with different licenses are to be included at build-time, their respective licenses might either be consistent or conflict. Further, if designed systems include

components with conflicting licenses, then one or more of the conflicting components must be excluded in the build-time release or must be abstracted behind an open API or middleware, with users required to download and install to enable the intended operation. (This is common in Linux distributions subject to GPL, where for example users may choose to acquire and install proprietary run-time components, like proprietary media players). So a component license conflict need not be a show-stopper if identified at design time. However, developers have to be able to determine which components’ licenses conflict and to take appropriate steps at design, build, and run times, consistent with the different concerns and requirements that apply at each phase [cf. Scacchi and Alspaugh 2008].

In order to fulfill our goals, we need a scheme for expressing software licenses that is more formal and less ambiguous than natural language, and that allows us to identify conflicts arising from the various rights and obligations pertaining to two or more component’s licenses. We considered relatively complex structures (such as Hohfeld’s eight fundamental jural relations [Hohfeld 1913]) but, applying Occam’s razor, selected a simpler structure. We start with a tuple $\langle actor, operation, action, object \rangle$ for expressing a right or obligation. The *actor* is the “licensee” for all the licenses we have examined. The *operation* is one of the following: “may”, “must”, or “must not”, with “may” expressing a right and “must” and “must not” expressing obligations; following Hohfeld, the lack of a right (which would be “may not”) correlates with a duty to not exercise the right (“must not”), and whenever lack of a right seemed significant in a license we expressed it as a negative obligation with “must not”. The *action* is a verb or verb phrase describing what may, must, or must not be done, with the *object* completing the description. We specify an object separately from the action in order to minimize the set of actions. A license then may be expressed as a set of rights, with each right associated (in that license) with zero or more obligations that must be fulfilled in order to enjoy that right. Figure 2 displays the tuples and associations for two of the rights and their associated obligations for the academic BSD software license. Note that the first right is granted without corresponding obligations.

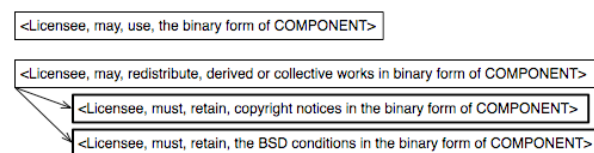


Figure 2. A portion of the BSD license tuples

We now turn to examine how OA software systems that include components with different licenses can be designed and analyzed while effectively tracking their rights and obligations.

When designing an OA software system, there are heuristics that can be employed to enable architectural design choices that might otherwise be excluded due to license conflicts. First, it is possible to employ a “license firewall” which serves to limit the scope of reciprocal obligations. Rather than simply interconnecting conflicting components through static linking of components at build time, such components can be logically connected via dynamic links, client-server protocols, license shims (e.g., via LGPL connectors), or runtime plug-ins. Second, the source code of statically linked OSS components must be made public. Third, it is necessary to include appropriate notices and publish required sources when academic licenses are employed. However, even using design heuristics such as these (and there are many), keeping track of license rights and obligations across components that are interconnected in complex OAs quickly become too cumbersome. Thus, automated support needs to be provided to help overcome and manage the multi-component, multi-license complexity.

5. Automating analysis of software license rights and obligation

We find that if we start from a formal specification of a software system’s architecture, then we can associate software license attributes with the system’s components, connectors, and sub-system architectures and calculate the copyright rights and obligations for the system. Accordingly, we employ an architectural description language specified in xADL [2005] to describe OAs that can be designed and analyzed with a software architecture design environment [Medvidovic 1999], such as ArchStudio4 [2006]. We have taken this environment and extended it with a Software Architecture License Traceability Analysis module [cf. Asuncion 2008]. This allows for the specification of licenses as a list of attributes (license tuples) using a form-based user interface, similar to those already used and known for ArchStudio4 and xADL [ArchStudio 2006, Medvidovic 1999].

Figure 3 shows a screenshot of an ArchStudio4 session in which we have modeled the OA seen in Figure 1. OA software components, each of which has an associated license, are indicated by darker shaded boxes. Light shaded boxes indicate connectors. Architectural connectors may or may not have associated license information; those with licenses (such as archi-

tectural connectors that represent functional code) are treated as components during license traceability analysis. A directed line segment indicates a link. Links connect interfaces between the components and connectors. Furthermore, the Mozilla component as shown here contains a hypothetical subarchitecture for modeling the role of intra-application scripting, as might be useful in specifying license constraints for Rich Internet Applications. This subarchitecture is specified in the same manner as the overall system architecture, and is visible in Figure 5. The automated environment allows for tracing and analysis of license attributes and conflicts.

Figure 4 shows a view of the internal XML representation of a software license. Analysis and calculations of rights, obligations, and conflicts for the OA are done in this form. This schematic representation is similar in spirit to that used for specifying and analyzing privacy and security regulations associated with certain software systems [Breaux and Anton 2008].

With this basis to build on, it is now possible to analyze the alignment of rights and obligations for the overall system:

1. Propagation of reciprocal obligations

Reciprocal obligations are imposed by the license of a GPL’d component on any other component that is part of the same “work based on the Program” (i.e. on the first component), as defined in GPL. We follow the widely-accepted interpretation that build-time static linkage propagate the reciprocal obligations, but the “license firewalls” do not. Analysis begins, therefore, by propagating these obligations along all connectors that are not license firewalls.

2. Obligation conflicts

An obligation can conflict with another obligation contrary to it, or with the set of available rights, by requiring a copyright right that has not been granted. For instance, the Corel proprietary license for the WordPerfect component, CTL (Corel Transactional License), may be taken to entail that a licensee must not redistribute source code. However, an OSS license, GPL, may state that a licensee must redistribute source code. Thus, the conflict appears in the modality of the two otherwise identical obligations, “must not” in CTL and “must” in GPL. A conflict on the same point could occur also between GPL and a component whose license fails to grant the right to distribute its source code.

This phase of the analysis is affected by the overall set of rights that are required. If conflicts arise involving the union of all obligations in all components' licenses, it may be possible to eliminate some conflicts by selecting a smaller set of rights, in which case only the obligations for those rights need be considered.

Figure 5 shows a screenshot in which the License Traceability Analysis module has identified obligation conflicts between the licenses of two pairs of components ("WordPerfect" and "Linux OS", and "GUIDisplayManager" and "GUIScriptInterpreter").

3. Rights and obligations calculations

The rights available for the entire system (use, copy, modify, etc.) then are calculated as the intersection of the sets of rights available for each component of the system.

The obligations required for the whole system then are the union of the specific obligations for each component that are associated with those rights. Examples of specific obligations are "Licensee must retain copyright notices in the binary form of `module.c`" or "Licensee must publish the source code of `component.java` version 1.2.3."

Figure 6 shows a report of the calculations for the hypothetical subarchitecture of the Mozilla component in our archetypal architecture, exhibiting an obligation conflict and the single copyright right (to run the system) that the prototype tool shows would be available for the subarchitecture as a whole if the conflict is resolved; a production tool would also list the rights (none) currently available.

If a conflict is found involving the obligations and rights of linked components, it is possible for the system architect to consider an alternative linking scheme, employing one or more connectors along the paths between the components that act as a license firewall, thereby mitigating or neutralizing the component-component license conflict. This means that the architecture and the environment together can determine what OA design best meets the problem at hand with available software components. Components with conflicting licenses do not need to be arbitrarily excluded, but instead may expand the range of possible architectural alternatives if the architect seeks such flexibility and choice.

At build-time (and later at run-time), many of the obligations can be tested and verified, for example that the binaries contain the appropriate notices for their li-

censes, and that the source files are present in the correct version on the Web. These tests can be generated from the internal list of obligations and run automatically. If the system's interface were extended to add a control for it, the tests could be run by a deployed system.

The prototype License Traceability Analysis module provides a proof-of-concept for this approach. We encoded the core provisions of four licenses in XML for the tool—GPL, MPL, CTL, and AFL (Academic Free License)—to examine the effectiveness of the license tuple encoding and the calculations based upon it. While it is clear that we could use a more complex and expressive structure for encoding licenses, in encoding the license provisions to date we found that the tuple representation was more expressive than needed; for example, the actor was always "licensee" and seems likely to remain so, and we found use for only three operations or modalities. At this writing, the module shows proof of concept for calculating with reciprocal obligations by propagating them to adjacent statically-linked modules; the extension to all paths not blocked by license firewalls is straightforward and is independent of the scheme and calculations described here. Reciprocal obligations are identified in the tool by lookup in a table, and the meaning and scope of reciprocity is hard-coded; this is not ideal, but we considered it acceptable since the legal definition in terms of the reciprocal licenses will not change frequently. We also focused on the design-time analysis and calculation rather than build- or run-time as it involves the widest range of issues, including representations, rights and obligations calculations, and design guidance derived from them.

Based on our analysis approach, it appears that the questions of what license (if any) covers a specific configured system, and what rights are available for the overall system (and what obligations are needed for them) are difficult to answer without automated license-architecture analysis. This is especially true if the system or sub-system is already in operational run-time form [cf. Kazman and Carrière 1999]. It might make distribution of a composite OA system somewhat problematic if people cannot understand what rights or obligations are associated with it. We offer the following considerations to help make this clear. For example, a Mozilla/Firefox Web browser covered by the MPL (or GPL or LGPL, in accordance with the Mozilla Tri-License) may download and run intra-application script code that is covered by a different license. If this script code is only invoked via dynamic run-time linkage, or via a client-server transaction protocol, then there is no propagation of license rights or obligations. However,

if the script code is integrated into the source code of the Web browser as persistent part of an application (e.g., as a plug-in), then it could be viewed as a configured sub-system that may need to be accessed for license transfer or conflict implications. Another different kind of example can be anticipated with application programs (like Web browsers, email clients, and word processors) that employ Rich Internet Applications or mashups entailing the use of content (e.g., textual character fonts or geographic maps) that is subject to copyright protection, if the content is embedded in and bundled with the scripted application sub-system. In such a case, the licenses involved may not be limited to OSS or proprietary software licenses.

In the end, it becomes clear that it is possible to automatically determine what rights or obligations are associated with a given system architecture at design-time, and whether it contains any license conflicts that might prevent proper access or use at build-time or run-time, given an approach such as ours.

6. Discussion

Software system configurations in OAs are intended to be adapted to incorporate new innovative software technologies that are not yet available. These system configurations will evolve and be refactored over time at ever increasing rates [Scacchi 2007], components will be patched and upgraded (perhaps with new license constraints), and inter-component connections will be rewired or remediated with new connector types. As such, sustaining the openness of a configured software system will become part of ongoing system support, analysis, and validation. This in turn may require ADLs to include OSS licensing properties on components, connectors, and overall system configuration, as well as in appropriate analysis tools [cf. Bass, Clements, and Kazman 2003, Medvidovic 1999].

Constructing these descriptions is an incremental addition to the development of the architectural design, or alternative architectural designs. But it is still time-consuming, and may present a somewhat daunting challenge for large pre-existing systems that were not originally modeled in our environment.

Advances in the identification and extraction of configured software elements at build time, and their restructuring into architectural descriptions is becoming an ever more automatable endeavor [cf. Choi 1990, Kazman 1999, Jansen 2008]. Further advances in such efforts have the potential to automatically produce architectural descriptions that can either be manually or semi-automatically annotated with their license con-

straints, and thus enable automated construction and assessment of build-time software system architectures.

The list of recognized OSS licenses is long and ever-growing, and as existing licenses are tested in the courts we can expect their interpretations to be clarified and perhaps altered; the GPL definition of “work based on the Program”, for example, may eventually be clarified in this way, possibly refining the scope of reciprocal obligations. Our expressions of license rights and obligations are for the most part compared for identical actors, actions, and objects, then by looking for “must not” in one and either “must” or “may” in the other, so that new licenses may be added by keeping equivalent rights or obligations expressed equivalently. Reciprocal obligations, however, are handled specially by hard-coded algorithms to traverse the scope of that obligation, so that addition of obligations with different scope, or the revision of the understanding of the scope of an existing obligation, requires development work. Possibly these issues will be clarified as we add more licenses to the tool and experiment with their application in OA contexts.

Lastly, our scheme for specifying software licenses offers the potential for the creation of shared repositories where these licenses can be accessed, studied, compared, modified, and redistributed.

7. Conclusion

The relationship between open architecture, open source software, and multiple software licenses is poorly understood. OSS is often viewed as primarily a source for low-cost/free software systems or software components. Thus, given the goal of realizing an OA strategy together with the use of OSS components and open APIs, it has been unclear how to best align software architecture, OSS, and software license regimes to achieve this goal. Subsequently, the central problem we examined in this paper was to identify principles of software architecture and software copyright licenses that facilitate or inhibit how best to insure the success of an OA strategy when OSS and open APIs are required or otherwise employed. In turn, we presented an analysis scheme and operational environment that demonstrates that an automated solution to this problem exists.

We have developed and demonstrated an operational environment that can automatically determine the overall license rights, obligations, and constraints associated with a configured system architecture whose components may have different software licenses. Such an environment requires the annotation of the participating software elements with their corresponding li-

censes. These annotated software architectural descriptions can be prescriptively analyzed at design-time as we have shown, or descriptively analyzed at build-time or run-time. Such a solution offers the potential for practical support in design-, build-, and run-time license conformance checking and the ever-more complex problem of developing large software systems from configurations of software elements that can evolve over time.

8. Acknowledgments

The research described in this report has been supported by grants #0808783 from the U.S. National Science Foundation, the Acquisition Research Program at the Naval Postgraduate School, and the Daegu Global R&D Collaboration Center, Daegu, Korea. No endorsement implied.

References

- Alspaugh, T.A and Antón, A.I., (2007). Scenario Support for Effective Requirements, Information and Software Technology, 50(3), 198-220.
- ArchStudio (2006). ArchStudio 4 Software and Systems Architecture Development Environment. Institute for Software Research, University of California, Irvine.
<http://www.isr.uci.edu/projects/archstudio/>
- Asuncion, H. (2008). Towards Practical Software Traceability, in Companion of the 30th Intern. Conf. Software Engineering, 1023-1026, Leipzig, Germany.
- Bass, L., Clements, P., and Kazman, R., (2003). Software Architecture in Practice, 2nd Edition, Addison-Wesley Professional, New York..
- Breaux, T.D. and Anton, A.I. (2008). Analyzing Regulatory Rules for Privacy and Security Requirements, IEEE Trans. Software Engineering, 34(1), 5-20.
- Choi, S. and Scacchi, W. (1990). Extracting and Restructuring the Design of Large Systems, IEEE Software, 7(1), 66-71.
- Feldt, K., (2007). Programming Firefox: Building Rich Internet Applications with XUL, O'Reilly Press, Sebastopol, CA.
- Fontana, R., Kuhn, B.M., Molgen, E., et al. (2008). A Legal Issues Primer for Open Source and Free Software Projects, Software Freedom Law Center, Version 1.5.1,
<http://www.softwarefreedom.org/resources/2008/foss-primer.pdf>
- Fielding, R. and Taylor, R.N., (2002). Principled Design of the Modern Web Architecture, ACM Transactions Internet Technology, 2(2), 115-150.
- Hohfeld, W.N. (1913). Some Fundamental Legal Conceptions as Applied in Judicial Reasoning. Yale Law Journal, 23(1), 16-59.
- Jansen, A., Bosch, J., and Avgeriou, P. (2008). Documenting After the Fact: Recovering Architectural Design Decisions, J. Systems and Software, 81(4), 536-557.
- Kazman, R. and Carrière, J. (1999). Playing Detective: Reconstructing Software Architecture from Available Evidence. J. Automated Software Engineering, 6(2), 107-138.
- Kuhl, F., Weatherly, R., and Dahmann, J., (2000). Creating Computer Simulation Systems: An Introduction to the High Level Architecture, Prentice-Hall PTR, Upper Saddle River, New Jersey.
- Medvidovic, N., Rosenblum, D.S., and Taylor, R.N. (1999). A Language and Environment for Architecture-Based Software Development and Evolution. In Proc. 21st Intern. Conf. Software Engineering (ICSE '99). 44-53, IEEE Computer Society. Los Angeles, CA.
- Meyers, B.C. and Obendorf, P., (2001). Managing Software Acquisition: Open Systems and COTS Products, Addison-Wesley, New York.
- Nelson L. and Churchill, E.F., (2006). Repurposing: Techniques for Reuse and Integration of Interactive Services, Proc. 2006 IEEE Intern. Conf. Information Reuse and Integration, September.
- OSI (2008). The Open Source Initiative,
<http://www.opensource.org/>
- Rosen, L. (2005). Open Source Licensing: Software Freedom and Intellectual Property Law, Prentice-Hall PTR, Upper Saddle River, New Jersey.
<http://www.rosenlaw.com/oslbook.htm>
- Scacchi, W., (2002). Understanding the Requirements for Developing Open Source Software Systems, IEE Proceedings--Software, 149(1), 24-39, February.
- Scacchi, W. (2007). Free/Open Source Software Development: Recent Research Results and Emerging Opportunities, Proc. European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, Dubrovnik, Croatia, 459-468.
- Scacchi, W. and Alspaugh, T.A. (2008). Emerging Issues in the Acquisition of Open Source Software within the U.S. Department of Defense, Proc. 5th Annual Acquisition Research Symposium, Vol. 1, 230-244, NPS-AM-08-036, Naval Postgraduate School, Monterey, CA
- St. Laurent, A.M., (2004). Understanding Open Source and Free Software Licensing, O'Reilly Press, Sebastopol, CA.

Ven, K. and Mannaert, H., (2008). Challenges and Strategies in the Use of Open Source Software by Independent Software Vendors, Information and Software Technology, 50, 991-1002.

Wheeler, D.A., (2007). Open Source Software (OSS) in U.S. Government Acquisitions, The DoD Software Tech News, 10(2), 7-13, June.

xADL (2005). xADL 2.0: Highly-extensible architecture description language for software and systems. Institute for Software Research, University of California, Irvine.
<http://www.isr.uci.edu/projects/xarchuci/>

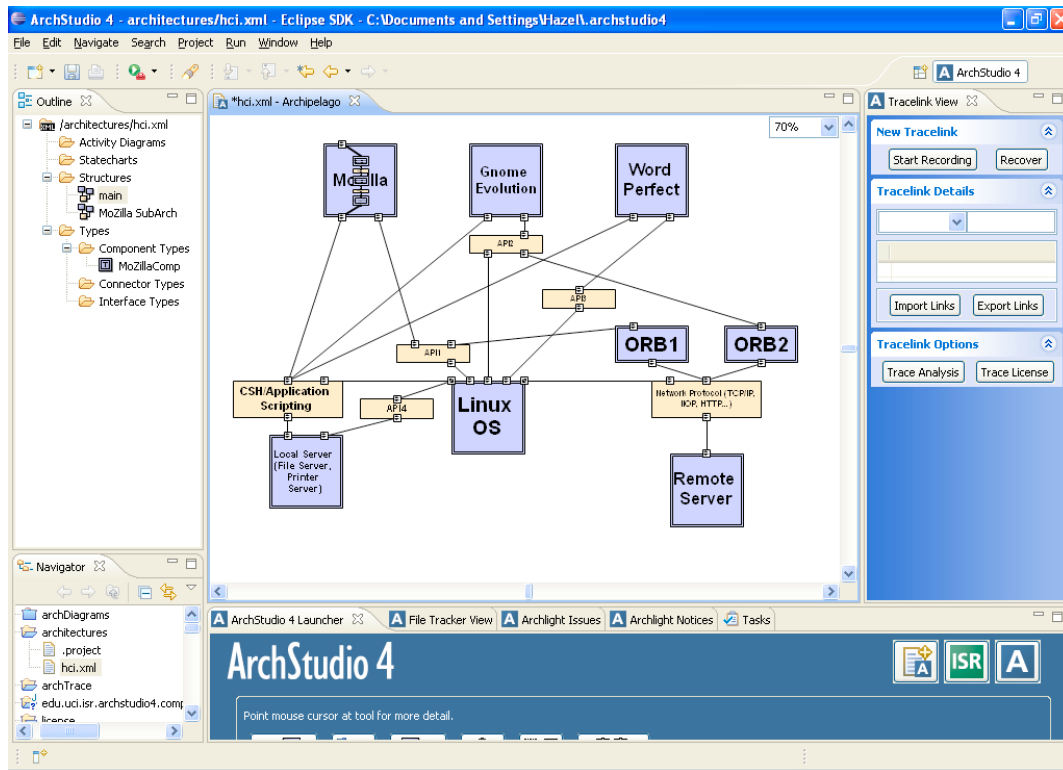


Figure 3. An ArchStudio 4 model of the open software architecture of Figure 1

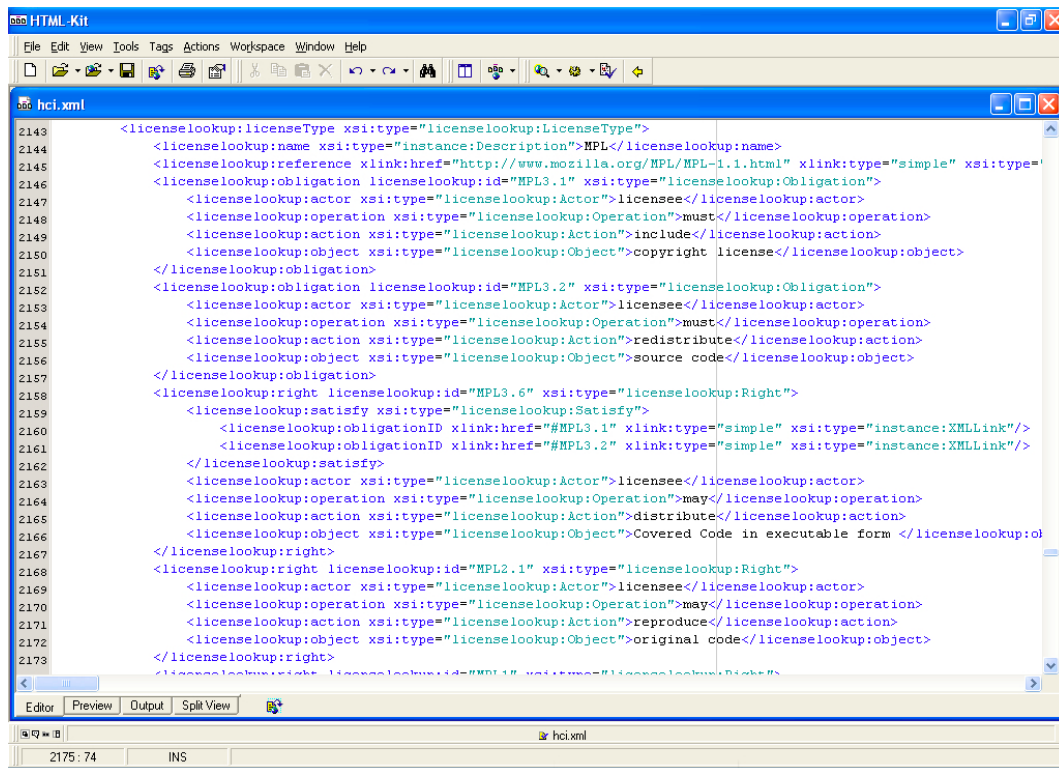


Figure 4. A view of the internal schematic representation of the Mozilla Public License

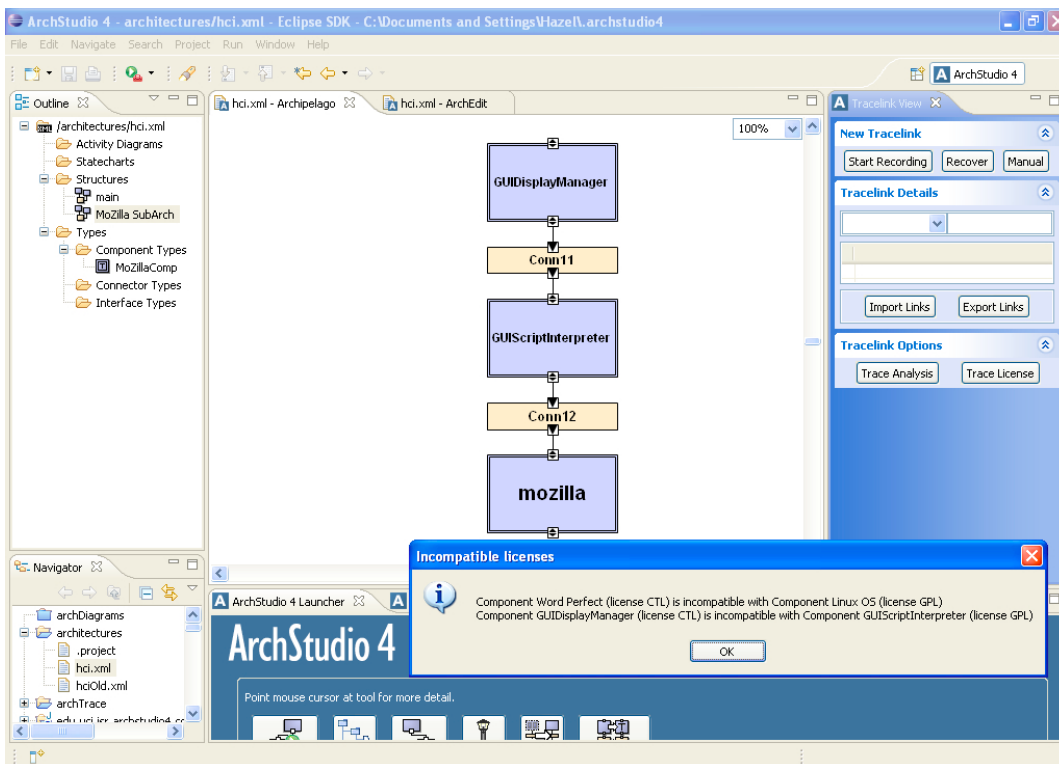
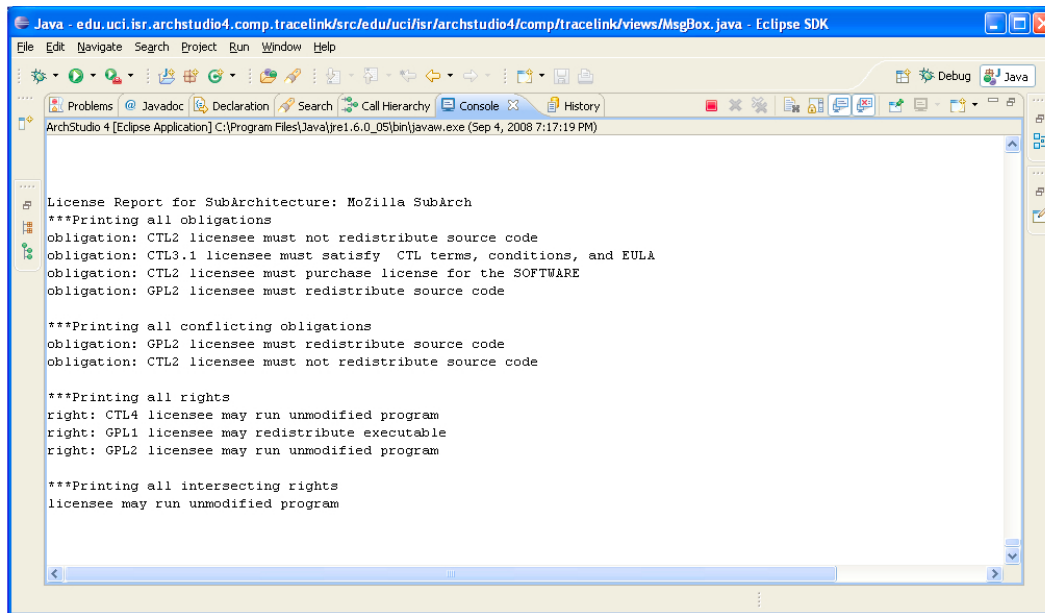


Figure 5. License conflicts have been identified between two pairs of components



The screenshot shows the Eclipse IDE's console window. The title bar reads "Java - edu.uci.isr.archstudio4.comp.tracelink/src/edu/uci/isr/archstudio4/comp/tracelink/views/MsgBox.java - Eclipse SDK". The console output is as follows:

```
License Report for SubArchitecture: Mozilla SubArch
***Printing all obligations
obligation: CTL2 licensee must not redistribute source code
obligation: CTL3.1 licensee must satisfy CTL terms, conditions, and EULA
obligation: CTL2 licensee must purchase license for the SOFTWARE
obligation: GPL2 licensee must redistribute source code

***Printing all conflicting obligations
obligation: GPL2 licensee must redistribute source code
obligation: CTL2 licensee must not redistribute source code

***Printing all rights
right: CTL4 licensee may run unmodified program
right: GPL1 licensee may redistribute executable
right: GPL2 licensee may run unmodified program

***Printing all intersecting rights
licensee may run unmodified program
```

Figure 6. A report identifying the obligations, conflicts, and rights for the architectural model

Understanding the Role of Licenses and Evolution in Open Architecture Software Ecosystems

Walt Scacchi

Institute for Software Research. University of California, Irvine USA

Thomas A. Alspaugh

Computer Science Dept., Georgetown University, Washington, DC, USA

Abstract

The role of software ecosystems in the development and evolution of heterogeneously-licensed open architecture systems has received insufficient consideration. Such systems are composed of components potentially under two or more licenses, open source or proprietary or both, in an architecture in which evolution can occur by evolving existing components, replacing them, or refactoring. The software licenses of the components both facilitate and constrain the system's ecosystem and its evolution, and the licenses' rights and obligations are crucial in producing an acceptable system. Consequently, software component licenses and the architectural composition of a system determine the software ecosystem niche where a system lies. Understanding and describing software ecosystem niches is a key contribution of this work. A case study of an open architecture software system that articulates different niches is employed to this end. We examine how the architecture and software component licenses of a composed system at design-time, build-time, and run-time help determine the system's software ecosystem niche and provide insight and guidance for identifying and selecting potential evolutionary paths of system, architecture, and niches.

Keywords: Software architecture, software ecosystems, software licenses, open

Email addresses: wscacchi@ics.uci.edu (Walt Scacchi), alspaugh@cs.georgetown.edu (Thomas A. Alspaugh)

1. Introduction

A substantial number of development organizations are adopting a strategy in which a software-intensive system (one in which software plays a crucial role) is developed with an *open architecture* (OA) [29], whose components may be open source software (OSS) or proprietary with open application programming interfaces (APIs). Such systems evolve not only through the evolution of their individual components, but also through replacement of one component by another, possibly from a different producer or under a different license. With this approach, another organization often comes between software component producers and system consumers. These organizations take on the role of system architect or integrator, either as independent software vendors, government contractors, system integration consultants, or in-house system integrators. In turn, such an integrator designs a system architecture that can be composed of components largely produced elsewhere, interconnected through interfaces accommodating use of dynamic links, intra- or inter-application scripts, communication protocols, software buses, databases/repositories, plug-ins, libraries or software shims as necessary to achieve the desired result. An OA development process results in an ecosystem in which the integrator is influenced from one direction by the goals, interfaces, license choices, and release cycles of the software component producers, and from another direction by the needs of the system's consumers. As a result the software components are reused more widely, and the resulting OA systems can achieve reuse benefits such as reduced costs, increased reliability, and potentially increased agility in evolving to meet changing needs. An emerging challenge is to realize the benefits of this approach when the individual components are *heterogeneously licensed* [2, 17, 33], each potentially with a different license, rather than a single OSS license as in uniformly-licensed OSS projects or a single proprietary license as in proprietary development.

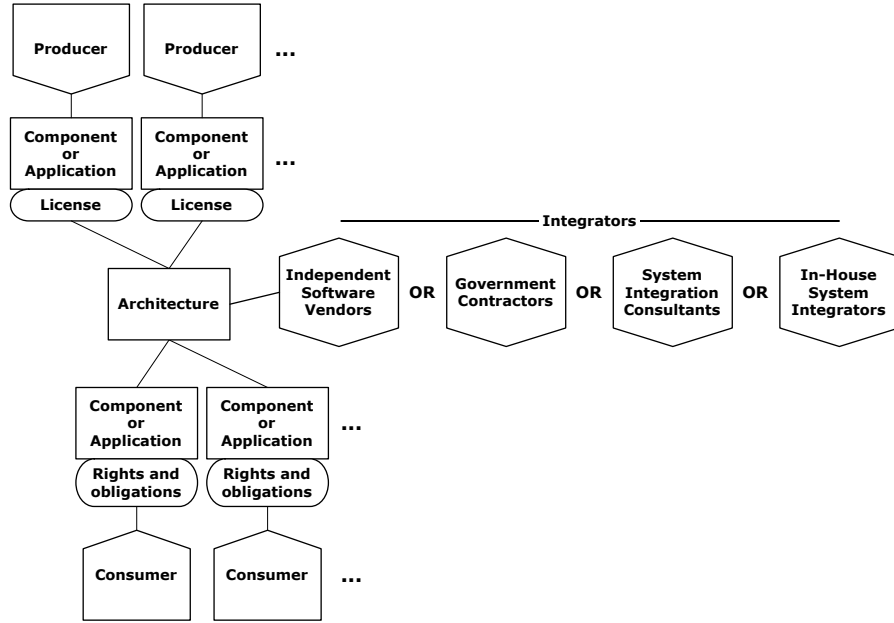


Figure 1: Schema for software supply networks

This challenge is inevitably entwined with the software ecosystems that arise for OA systems (Figure 1). We find that an OA software ecosystem involves organizations and individuals producing, composing, and consuming components that articulate software supply networks from producers to consumers, but also

- the OA of the system(s) in question,
- the open interfaces met by the components,
- the degree of coupling in the evolution of related components, and
- the rights and obligations resulting from the software licenses under which various components are released, that propagate from producers to consumers.

A motivating example of this approach is the Unity game development tool, produced by Unity Technologies [35]. Its license agreement, from which we quote below, lists eleven distinct licenses and indicates the tool is produced, apparently

using an OA approach, using at least 18 externally produced components or groups of components:

1. The Mono Class Library, Copyright 2005-2008 Novell, Inc.
2. The Mono Runtime Libraries, Copyright 2005-2008 Novell, Inc.
3. Boo, Copyright 2003-2008 Rodrigo B. Oliveira
4. UnityScript, Copyright 2005-2008 Rodrigo B. Oliveira
5. OpenAL cross platform audio library, Copyright 1999-2006 by authors.
6. PhysX physics library. Copyright 2003-2008 by Ageia Technologies, Inc.
7. libvorbis. Copyright (c) 2002-2007 Xiph.org Foundation
8. libtheora. Copyright (c) 2002-2007 Xiph.org Foundation
9. zlib general purpose compression library. Copyright (c) 1995-2005 Jean-loup Gailly and Mark Adler
10. libpng PNG reference library
11. jpeglib JPEG library. Copyright (C) 1991-1998, Thomas G. Lane.
12. Twilight Prophecy SDK, a multi-platform development system for virtual reality and multimedia. Copyright 1997-2003 Twilight 3D Finland Oy Ltd
13. dynamic_bitset, Copyright Chuck Allison and Jeremy Siek 2001-2002.
14. The Mono C# Compiler and Tools, Copyright 2005-2008 Novell, Inc.
15. libcurl. Copyright (c) 1996-2008, Daniel Stenberg <daniel@haxx.se>.
16. PostgreSQL Database Management System
17. FreeType. Copyright (c) 2007 The FreeType Project (www.freetype.org).
18. NVIDIA Cg. Copyright (c) 2002-2008 NVIDIA Corp.

The software ecosystem for Unity3D as a standalone software package is delimited by the diverse set of software components listed above. However the architecture that integrates or composes these components is closed and thus unknown to a system consumer, as is the manner in which the different licenses associated with these components impose obligations or provide rights to consumers, or on the other components to which they are interconnected. Subsequently, we see that software ecosystems can be understood in part by examining relationships between architectural composition of software components that are subject to different licenses, and this necessitates access to the system's architecture composition. By examining the open architecture of a specific composed software system, it becomes possible to explicitly identify the software ecosystem niche in which the system is embedded.

A software ecosystem constitutes a software supply network that connects software producers to integrators to consumers through licensed components

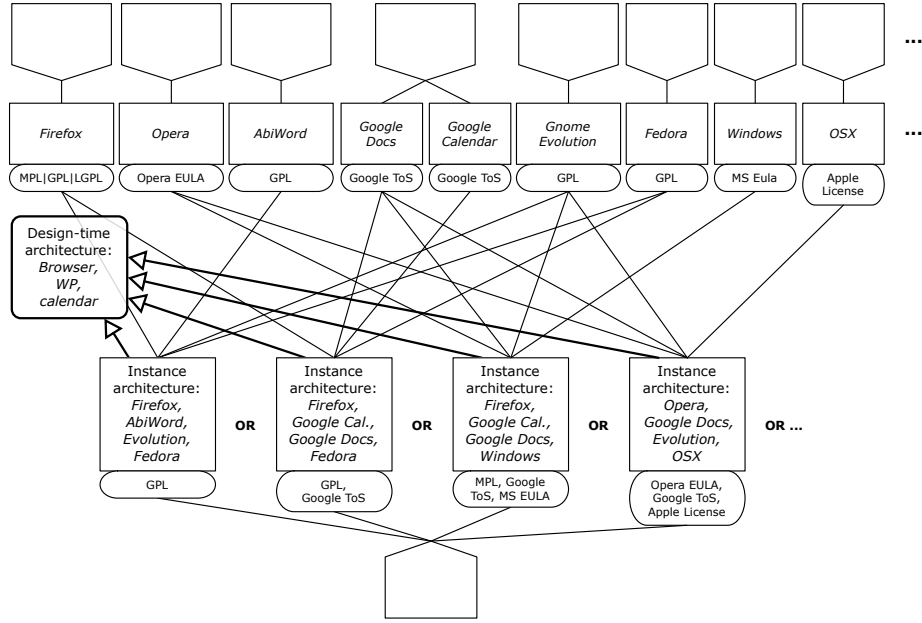


Figure 2: Ecosystem for three possible instantiations of a single design architecture

and composed systems. Figure 2 outlines the software ecosystem elements and relationships for an OA case study that we examine in this paper.

A *software ecosystem niche* articulates a specific software supply network that interconnects particular software producers, integrators, and consumers. A software system defined niche may lie within an existing single ecosystem, or one that may span a network of software producer ecosystems. A composed software system architecture helps determine the software ecosystem niche, since the architecture identifies the components, their licenses and producers, and thus the network of software ecosystems in which it participates. Such a niche also transmits license-borne obligations and access/usage rights passed from the participating software component producers through integrators and to system consumers. Thus, system architects or component integrators help determine in which software ecosystem niche a given instance architecture for the system participates.

As a software system evolves over time, as its components are updated or

changed, or their architectural interconnections are refactored, it is desirable to determine whether and how the system’s ecosystem niche may have changed. A system may evolve because its consumers want to migrate to alternatives from different component producers, or choose components whose licenses are more/less desirable. Software system consumers may want to direct their system integrators to compose the system’s architecture so as to move into or away from certain niches. Consequently, system integrators can update or modify system architectural choices at design-time, build-time (when components are compiled together into an executable), or run-time (when bindings to remote executable services are instantiated) to move a software system from one niche to another. Thus, understanding how software ecosystem niches emerge is a useful concept that links software engineering concerns for software architecture, system integration/composition, and software evolution to organizational and supply network relationships between software component producers, integrators and system consumers.

To help explain how OA systems articulate software ecosystem niches, we provide a software architecture case study for use in this paper. This system is intentionally simple for expository purposes. Its architectural design composes a web browser, word processor, calendaring and email applications, host platform operating system, and remote services as its components. Equivalent components from different OSS or proprietary software producers can be identified, where each alternative is subject to a different type of software license. For example, for Web browsers, we consider the Firefox browser from the Mozilla Foundation, which comes with a choice of OSS license (MPL, GPL, or LGPL), and the Opera browser from Opera Software, which comes with a proprietary software end-user license agreement (EULA). Similarly, for word processor, we consider the OSS AbiWord application (GPL) and Web-based Google Docs service (proprietary Terms of Service). The OA we describe covers a number of systems we have identified, built, and deployed in a university research laboratory. Example niches involving these components appear in Figures 3 and 4. We have also developed OA systems with more complex architectures that in-

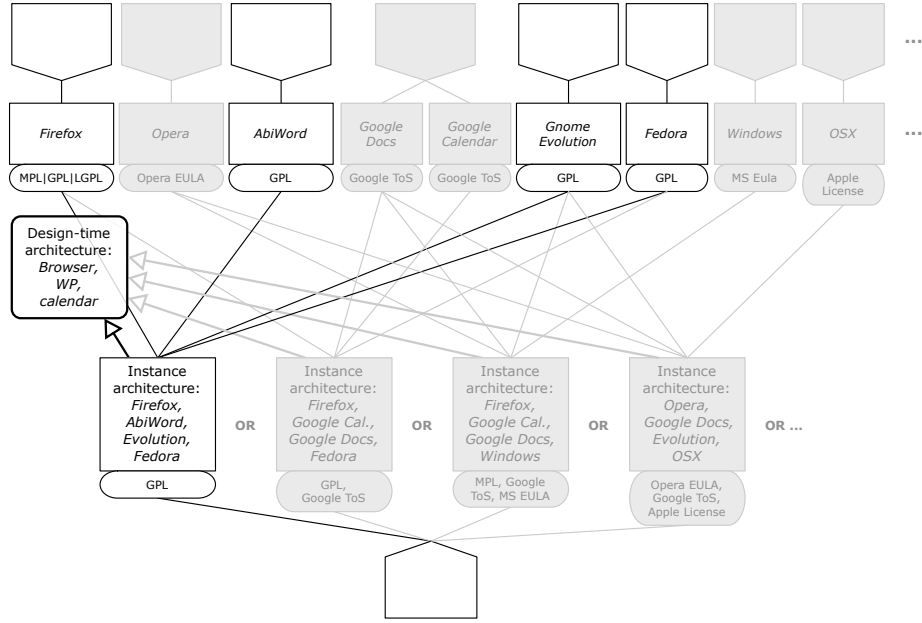


Figure 3: The ecosystem niche for one instance architecture

corporate components for content management systems (Drupal), wikis (MediaWiki), blogs (B2evolution), teleconferencing and media servers (Flash media server, Red5 media server), chat (BlaB! Lite), Web spiders and search engines (Nutch, Lucene, Sphider), relational database management systems (MySQL), and others. Furthermore, the OSS application stacks and infrastructure (platform) stacks found at BitNami.org/stacks (accessed 29 April 2010) could also be incorporated in OA systems, as could their proprietary counterparts. However, these more complex OAs still reflect the core architectural concepts and constructs, as well as software ecosystem relationships, that we present in our example case study in a simpler manner.

The software ecosystem niches for the case study system, or indeed any system, depend on which component implementations are used and the architecture in which they are combined and instantiated, as does the overall rights and obligations for the instantiated system. In addition, we build on previous work on heterogeneously-licensed systems [2, 17, 33] by examining how OA development

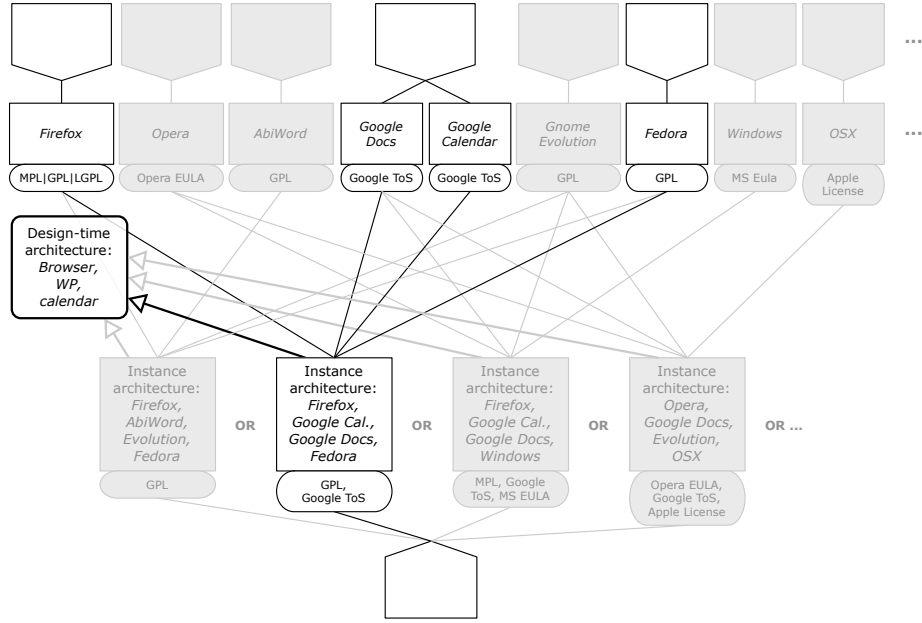


Figure 4: The ecosystem niche for a second instance architecture

affects and is affected by software ecosystems, and the role of component licenses in shaping OA software ecosystem niches.

Consequently, we focus our attention to understand the ecosystem of an open architecture software system such that:

- It must rest on a license structure of rights and obligations (Section 4), focusing on obligations that are enactable and testable¹.
- It must take account of the distinctions between the design-time, build-time, and distribution-time architectures (Section 3) and the rights and obligations that come into play for each of them.

¹For example, many OSS licenses include an obligation to make a component's modified code public, and whether a specific version of the code is public at a specified Web address is both enactable (it can be put into practice) and testable. In contrast, the General Public License (GPL) v.3 provision "No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty" is not enactable in any obvious way, nor is it testable — how can one verify what others deem?

- It must distinguish the architectural constructs significant for software licenses, and embody their effects on rights and obligations (Section 3).
- It must define license architectures.
- It must account for alternative ways in which software systems, components, and licenses can evolve (Section 5), and
- It must provide an automated environment for creating and managing license architectures. We are developing a prototype that manages a license architecture as a view of the system architecture [3].

The remainder of this paper is organized as follows. Section 2 places this work in the context of related research. Section 3 discusses open architecture, and the influence of software licenses is discussed in Section 4. Section 5 addresses evolution of software ecosystems. Section 6 discusses some implications that follow from this study, and Section 7 concludes the paper.

2. Related Research

The study of software ecosystems is emerging as an exciting new area of systematic investigation and conceptual development within software engineering. Understanding the many possible roles that software ecosystems can play in shaping software engineering practice is gaining more attention since the concept first appeared [23]. Bosch [7] builds a conceptual lineage from software product line (SPL) concepts and practices [6, 13] to software ecosystems. SPLs focus on the centralized development of families of related systems from reusable components hosted on a common platform with an intra-organizational base, with the resulting systems either intended for in-house use or commercial deployments. Software ecosystems then are seen to extend this practice to systems hosted on an inter-organizational base, which may resemble development approaches conceived for virtual enterprises for software development [26]. Producers of commercial software applications or packages thus need to adapt their development strategy and business model to one focused on coordinating and

guiding decentralized software development of its products and enhancements (e.g., plug-in components).

Along with other colleagues [8, 12, 36], Bosch identifies alternative ways to connect reusable software components through integration and tight coupling found in SPLs, or via loose coupling using glue code, scripting or other late binding composition schemes in ecosystems or other decentralized enterprises [26, 27], as a key facet that can enable software producers to build systems from diverse sources.

Jansen and colleagues [18, 19] draw attention to their observation that software ecosystems (a) embed software supply networks that span multiple organizations, and (b) are embedded within a network of intersecting or overlapping software ecosystems that span the world of software engineering practice. Scacchi [32] for example, identifies that the world of open source software (OSS) development is a software ecosystem different from those of commercial software producers, and its supply networks are articulated within a network of FOSS development projects. Networks of OSS ecosystems have also begun to appear around very large OSS projects for Web browsers, Web servers, word processors, and others, as well as related application development environments like NetBeans and Eclipse, and these networks have become part of global information infrastructures [20].

OSS ecosystems also exhibit strong relationships between the ongoing evolution of OSS systems and their developer/user communities, such that the success of one co-depends on the success of the other [32]. Ven and Mannaert discuss the challenges independent software vendors face in combining OSS and proprietary components, with emphasis on how OSS components evolve and are maintained in this context [37].

Boucharas and colleagues [9] then draw attention to the need to more systematically and formally model the contours of software supply networks, ecosystems, and networks of ecosystems. Such a formal modeling base may then help in systematically reasoning about what kinds of relationships or strategies may arise within a software ecosystem. For example, Kuehnel [21] exam-

ines how Microsoft’s software ecosystem developed around in operating systems (MS Windows) and key applications (e.g., MS Office) may be transforming from “predator” to “prey” in its effort to control the expansion of its markets to accommodate OSS (as the extant prey) that eschew closed source software with proprietary software licenses.

Our work in this area builds on these efforts in the following ways. First, we share the view of a need for examining software ecosystems, but we start from software system architectures that can be formally modeled and analyzed with automated tool support [6, 34]. Explicit modeling of software architectures enables the ability to view and analyze them at design-time, build-time, or deployment/run-time. Software architectures also serve as a mechanism for coordinating decentralized software development across multi-site projects [30]. Similarly, explicit models allow for the specification of system architectures using either proprietary software components with open APIs, OSS components, or combinations thereof, thereby realizing open architecture (OA) systems [33]. We then find value in attributing open architecture components with their (intellectual property) licenses [3], since software licenses are an expression of contractual/social obligations that software consumers must fulfill in order to realize the rights to use the software in specified allowable manners, as determined by the software’s producers.

3. Open Architectures

Open architecture (OA) software is a customization technique introduced by Oreizy [29] that enables third parties to modify a software system through its exposed architecture, evolving the system by replacing its components. Increasingly more software-intensive systems are developed using an OA strategy, not only with OSS components but also proprietary components with open APIs (e.g. [35]). Using this approach can lower development costs and increase reliability and function [33]. Composing a system with heterogeneously-licensed components, however, increases the likelihood of conflicts, liabilities, and no-rights stemming from incompatible licenses. Thus, in our work we define an

OA system as a *software system consisting of components that are either open source or proprietary with open API, whose overall system rights at a minimum allow its use and redistribution, in full or in part.*

It may appear that using a system architecture that incorporate OSS components and uses open APIs will result in an OA system. But not all such architectures will produce an OA, since the (possibly empty) set of available license rights for an OA system depends on: (a) how and why OSS and open APIs are located within the system architecture, (b) how OSS and open APIs are implemented, embedded, or interconnected, and (c) the degree to which the licenses of different OSS components encumber all or part of a software system’s architecture into which they are integrated [1, 33].

The following kinds of software elements appearing in common software architectures can affect whether the resulting systems are open or closed [5].

Software source code components—These can be either (a) standalone programs, (b) libraries, frameworks, or middleware, (c) inter-application script code such as C shell scripts, or (d) intra-application script code, as for creating Rich Internet Applications using domain-specific languages such as XUL for the Firefox Web browser [15] or “mashups” [25]. Their source code is available and they can be rebuilt. Each may have its own distinct license, though often script code that merely connects programs and data flows does not, unless the code is substantial, reusable, or proprietary.

Executable components—These components are in binary form, and the source code may not be open for access, review, modification, or possible redistribution [31]. If proprietary, they often cannot be redistributed, and so such components will be present in the design- and run-time architectures but not in the distribution-time architecture.

Software services—An appropriate software service can replace a source code or executable component.

Application programming interfaces/APIs—Availability of externally visible and accessible APIs is the minimum requirement for an “open system” [24]. Open APIs are not and cannot be licensed, but they can limit the propa-

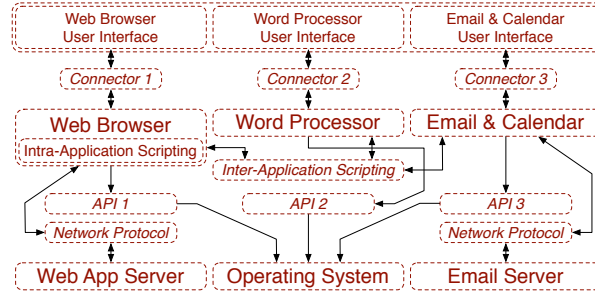


Figure 5: A design-time architecture

gation of license obligations.

Software connectors—Software whose intended purpose is to provide a standard or reusable way of communication through common interfaces, e.g. High Level Architecture [22], CORBA, MS .NET, Enterprise Java Beans, and GNU Lesser General Public License (LGPL) libraries. Connectors can also limit the propagation of license obligations.

Methods of composition—These include linking as part of a configured subsystem, dynamic linking, and client-server connections. Methods of composition affect license obligation propagation, with different methods affecting different licenses.

Configured system or subsystem architectures—These are software systems that are used as atomic components of a larger system, and whose internal architecture may comprise components with different licenses, affecting the overall system license. To minimize license interaction, a configured system or sub-architecture may be surrounded by what we term a *license firewall*, namely a layer of dynamic links, client-server connections, license shims, or other connectors that block the propagation of reciprocal obligations.

With these architectural elements, we can create an design-time or reference architecture for a system that conforms to the software supply network shown in Figure 2. This design-time architecture appears in Figure 5; note that it only specifies components by type rather than by producer, meaning the choice of producer component remains unbound at this point.

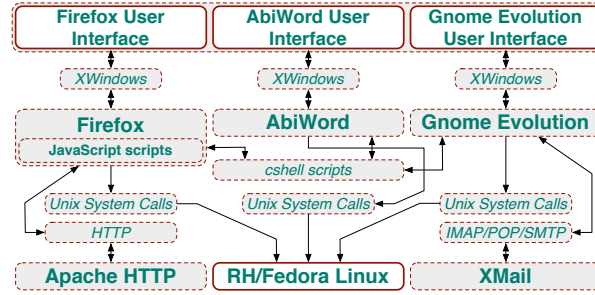


Figure 6: A build-time architecture

Then in Figure 6, we create a build-time rendering of this architectural design by selecting specific components from designated software producers. The gray boxes correspond to components and connectors not visible in the run-time instantiation of the system in Figure 7.

Figures 7, 8, and 9 display alternative run-time instantiations of the design-time architecture of Figure 5. The architectural run-time instance in Figure 7 corresponds to the software ecosystem niche shown in Figure 3; Figure 8 corresponds to the niche in Figure 4; and Figure 9 designates yet another niche different from the previous two.

Each component selection implies acceptance of the license obligations and rights that the producer seeks to transmit to the components consumers. However in an OA design development, component interconnections may be used to intentionally or unintentionally propagate these obligations onto other components whose licenses may conflict with them or fail to match [3, 17]; the system integrator can decide to insert software shims using scripts, dynamic links to remote services, data communication protocols, or libraries to mitigate or firewall the extent to which a component’s license obligations propagate. This style of build-time composition can be used to accommodate a system’s consumers’ policy for choosing systems that avoid certain licenses, or that isolate the license obligations of certain desirable components. It also allows system integrators and consumers to follow a “best of breed” policy in the selection of system components. Finally, if no license conflicts exist in the system, or if the integra-

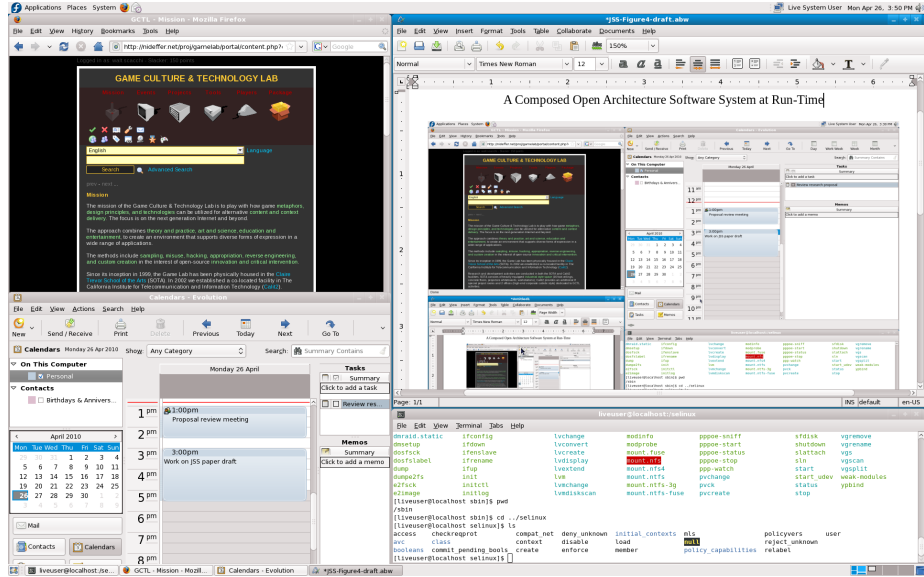


Figure 7: An instantiation at run-time (Firefox, AbiWord, Gnome Evolution, Fedora) of the build-time architecture of Figure 6 that determines the ecosystem niche of Figure 3

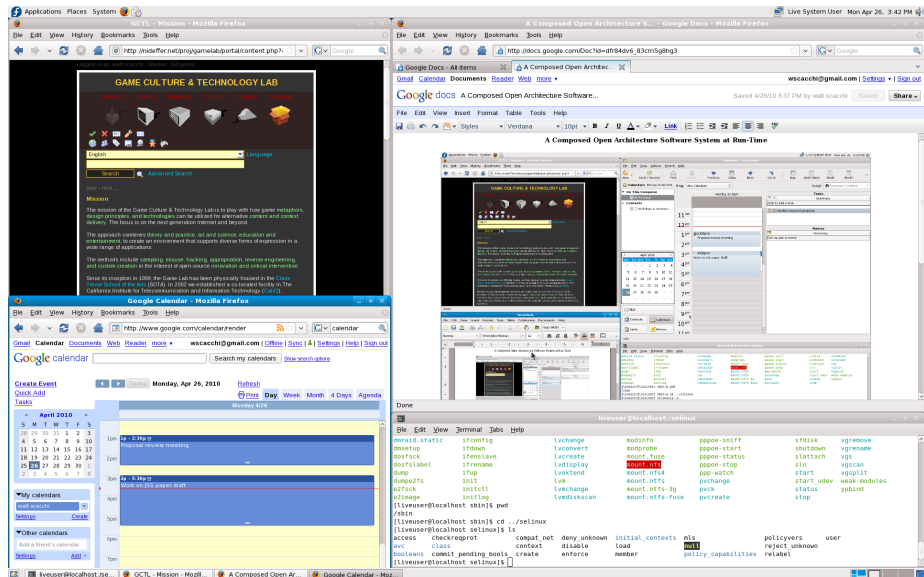


Figure 8: A second instantiation at run-time (Firefox, Google Docs and Calendar, Fedora) determining the ecosystem niche of Figure 4

and license notices.

Reciprocal OSS licenses take a more active stance towards sharing and reusing software by imposing the obligation that reciprocally-licensed software not be combined (for various definitions of “combined”) with any software that is not in turn also released under the reciprocal license. Those for which most or all ways of combining software propagate reciprocal obligations are termed *strongly reciprocal*. Examples are the GPL and the Affero GPL (AGPL). The purpose of the AGPL is to prevent a GPL software component from being integrated into an OA system as a remote server, or from being wrapped with shims to inhibit its ability to propagate the GPL obligations and rights. The purpose of these licenses is to ensure that software so licensed will maintain (and can propagate) the freedom to access, study, modify, and redistribute the software source code, which academic licenses do not. This in turn assures the access, use, and reusability of the source code for other software producers and system integrators. Those licenses for which only certain ways of combining software propagate reciprocal obligations are termed *weakly reciprocal*. Examples are the Lesser GPL (LGPL), Mozilla Public License (MPL), and Common Public License. The goals of reciprocal licensing are to increase the domain of OSS by encouraging developers to bring more components under its aegis, and to prevent improvements to OSS components from vanishing behind proprietary licenses.

Both proprietary and OSS licenses typically disclaim liability, assert no warranty is implied, and obligate licensees to not use the licensor’s name or trademarks. Newer licenses often cover patent issues as well, either giving a restricted patent license or explicitly excluding patent rights. The Mozilla Disjunctive Tri-License licenses the core Mozilla components under any one of three licenses (MPL, GPL, or the GNU Lesser General Public License LGPL); OSS developers or OA system integrators can choose the one that best suits their needs for a particular project and component.

The Open Source Initiative (OSI) maintains a widely-respected definition of “open source” and gives its approval to licenses that meet it [28]. OSI maintains

and publishes a repository of approximately 70 approved OSS licenses which tend to vary in the terms and conditions of their declared obligations and rights. However, all these licenses tend to cluster into either a strongly reciprocal, weakly reciprocal, or minimally restrictive/academic license type.

Common practice has been for an OSS project to choose a single license under which all its products are released, and to require developers to contribute their work only under conditions compatible with that license. For example, the Apache Contributor License Agreement grants enough of each author’s rights to the Apache Software Foundation for the foundation to license the resulting systems under the Apache Software License. This sort of rights regime, in which the rights to a system’s components are homogeneously granted and the system has a single well-defined OSS license, was the norm in the early days of OSS and continues to be practiced.

We have developed an approach for expressing software licenses that is more formal and less ambiguous than natural language, and that allows us to calculate and identify conflicts arising from the rights and obligations of two or more component’s licenses. Our approach is based on Hohfeld’s classic group of eight fundamental jural relations [Hohfeld 1913], of which we use right, duty, no-right, and privilege (Figure 10). We start with a tuple `<actor, operation, action, object>` for expressing a right or obligation. The actor is the “licensee” for all the licenses we have examined. The operation is one of the following: “may”, “must”, “must not”, or “need not”, with “may” and “need not” expressing rights and “must” and “must not” expressing obligations. Because copyright rights are only available to entities who have been granted a sublicense, only the listed rights are available, and the absence of a right means that it is not available. The action is a verb or verb phrase describing what may, must, must not, or need not be done, with the object completing the description. A license may be expressed as a set of rights, with each right associated with zero or more obligations that must be fulfilled in order to enjoy that right. Figure 11 sketches two rights from GPL 2.0, the first one with no obligations and the second with three corresponding obligations. Elsewhere, the details of this license specification

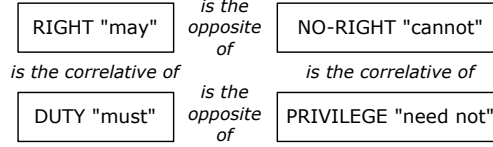


Figure 10: Hohfeld's four basic relations

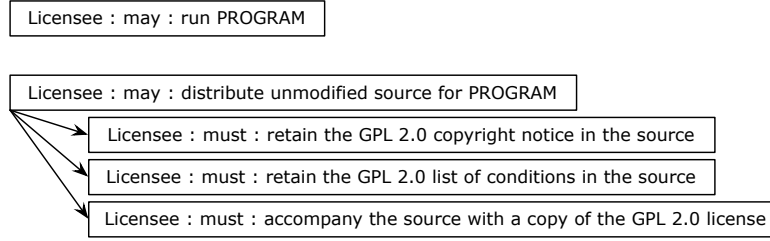


Figure 11: Tuples for some rights and obligations of the GPL 2.0 license

scheme, how it can be used to attribute software architectures to produce a system license architecture, and how it can be formalized into a semantic meta-model [3].

HLS designers have developed a number heuristics to guide architectural design while avoiding some license conflicts. First, it is possible to use a reciprocally-licensed component through a license firewall that limits the scope of reciprocal obligations. Rather than connecting conflicting components directly through static or other build-time links, the connection is made through a dynamic link, client-server protocol, license shim (such as an LGPL connector), or run-time plug-ins. A second approach used by a number of large organizations is simply to avoid using any components with reciprocal licenses. A third approach is to meet the license obligations (if that is possible) by for example retaining copyright and license notices in the source and publishing the source code. However, even using design heuristics such as these (and there are many), keeping track of license rights and obligations across components that are interconnected in complex OAs quickly becomes too cumbersome. Automated support is needed to manage the multi-component, multi-license complexity. Accordingly, we are developing an automated support capability as part of the

ArchStudio architecture design environment [14] that can analyze the addition of software license properties such as those shown in Figure 11 to the interfaces of software components in an OA system. For example, in Figure 12 we see a rendering of the OA system from our case study with the AbiWord word processing component highlighted. This component’s APIs would be attributed with the GPL license obligations and rights in Figure 11, since AbiWord is licensed under GPL, as are other components like the Gnome Evolution calendaring and email application and also the Red Hat/Fedora Linux operating system platform. As the architectural interconnections shown in the model of Figure 12 indicate that none of these components covered by GPL are directly interconnected to another licensed component, their license obligations do not propagate or become “viral” in this architectural composition. Replacing any of these GPL components with non-GPL but still OSS components would not change the total set of obligations and rights on the system with this architecture; the system would remain OSS, but the software ecosystem niche in which it resides would shift to another niche. Once again, software licenses interact with software architectures and together they help determine which software ecosystem niche will embed an instantiated run-time version of the system.

5. Architecture, License, and Ecosystem Evolution

An OA system can evolve by a number of distinct mechanisms, some of which are common to all systems but others of which are a result of heterogeneous component licenses in a single system.

By component evolution— One or more components can evolve, altering the overall system’s characteristics (for example, upgrading and replacing the Firefox Web browser from version 3.5 to 3.6). Such minor versions changes generally have no effect on system architecture.

By component replacement— One or more components may be replaced by others with modestly different functionality but similar interface, or with a different interface and the addition of shim code to make it match (for example, replacing the AbiWord word processor with either Open Office Writer or MS

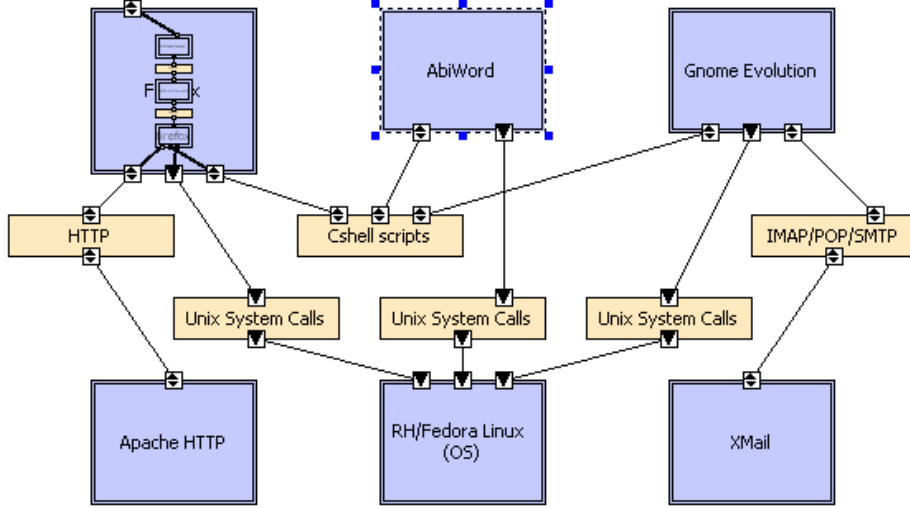


Figure 12: The model of the system architecture used in our case study as rendered in our automated tool, described elsewhere [2, 3]

Word). However, changes in the format or structure of component APIs may necessitate build-time and run-time updates to component connectors. Figure 13 shows some possible alternative system compositions that result from replacing components by others of the same type but with a different license.

By architecture evolution— The OA can evolve, using the same components but in a different configuration, altering the system characteristics. For example, as discussed in Section 4, revising or refactoring the configuration in which a component is connected can change how its license affects the rights and obligations for the overall system. This could arise when replacing word processing, calendaring, and email components and their connectors with Web-browser-based services such as Google Docs, Google Calendar, and Google Mail. The replacement would eliminate the legacy components and relocate the desired application functionality to operate within the Web browser component, resulting in what might be considered a simpler and easier-to-maintain system architecture, but one that is less open and now subject to a proprietary Terms of Service license. System consumer policy preferences for licenses, and subsequent participation in a different ecosystem niche, may thus mediate whether

such an alternative system architecture is desirable or not.

By component license evolution— The license under which a component is available may change, as for example when the license for the Mozilla core components was changed from the Mozilla Public License (MPL) to the current Mozilla Disjunctive Tri-License; or the component may be made available under a new version of the same license, as for example when the GNU General Public License (GPL) version 3 was released. The three architectures in Figure 13 that incorporate the Firefox Web browser show how its tri-license creates new evolutionary paths by offering different licensing options. These options and paths were not available previously with earlier versions of this component offered under only one or two license alternatives.

By a change to the desired rights or acceptable obligations— The OA system’s integrator or consumers may desire additional license rights (for example the right to sublicense in addition to the right to distribute), or no longer desire specific rights; or the set of license obligations they find acceptable may change. In either case the OA system evolves, whether by changing components, evolving the architecture, or other means, to provide the desired rights within the scope of the acceptable obligations. For example, they may no longer be willing or able to provide the source code for components within the reciprocity scope of a GPL-licensed module. Figure 14 shows an array of choices among types of licenses for different types of components that appear in the OA case study example. Each choice determines the obligations that component producers can demand of their consumers in exchange for the access/usage rights they offer

The interdependence of producers, integrators, and consumers results in a co-evolution of software systems and social networks within an OA ecosystem [32]. Closely-coupled components from different producers must evolve in parallel in order for each to provide its services, as evolution in one will typically require a matching evolution in the other. Producers may manage their evolution with a loose coordination among releases, for example as between the Gnome and Mozilla organizations. Each release of a producer component create a tension

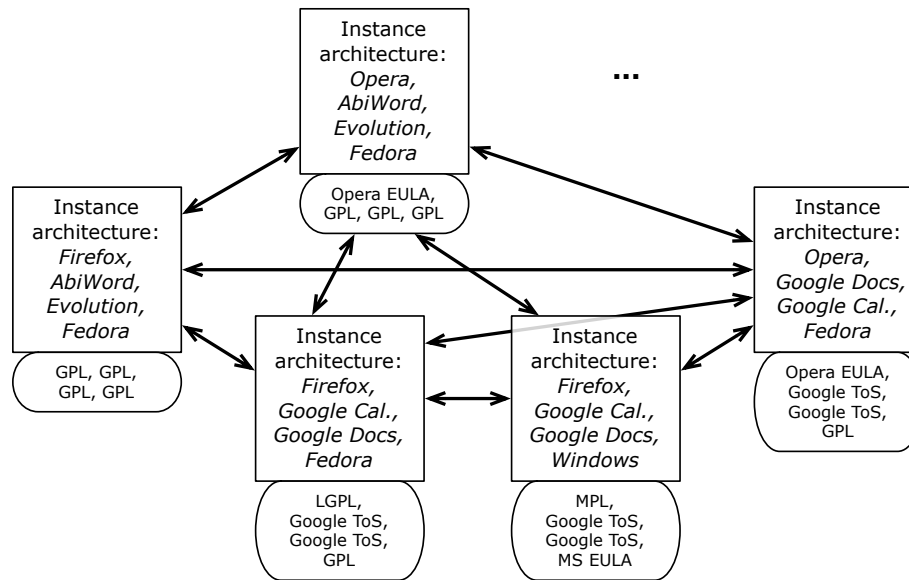


Figure 13: Possible evolutionary paths among a few instance architectures; some paths are impractical due to the changes in license obligations

	Browser	Word processor	Calendar, email	Platform
Proprietary	Opera (Opera EULA)	WordPerfect (Corel License)		Windows (MS EULA)
Strongly Reciprocal	Firefox (MPL or LGPL or GPL)	AbiWord (GPL)	Gnome Evolution (GPL)	Fedora (GPL)
Weakly Reciprocal or Academic		OpenOffice (LGPL)		FreeBSD (BSD variant)
Service		Google Docs (Google ToS)	Google Calendar (Google ToS)	

Figure 14: Some architecture choices and their license categories

through the ecosystem relationships with consumers and their releases of OA systems using those components, as integrators accommodate the choices of available, supported components with their own goals and needs. As discussed in our previous work [3], license rights and obligations are manifested at each component interface then mediated through the OA of the system to entail the rights and corresponding obligations for the system as a whole. As a result, integrators must frequently re-evaluate the OA system rights and obligations. In contrast to homogeneously-licensed systems, license change across versions is a characteristic of OA ecosystems, and architects of OA systems require tool support for managing the ongoing licensing changes.

6. Discussion

At least two topics merit discussion following from our approach to understanding of software ecosystems and ecosystem niches for OA systems: first, how might our results shed light on software systems whose architectures articulate a software product line; and second, what insights might we gain based on the results presented here on possible software license architectures for mobile computing ecosystems. Each is addressed in turn.

Software product lines (SPLs) rely on the development and use of explicit software architectures [6, 13]. However, the architecture of an SPL or software ecosystem does not necessarily require an OA—there is no need for it to be open. Thus, we are interested in discussing what happens when SPLs may conform to an OA, and to an OA that may be subject to heterogeneously licensed SPL components. Three considerations come to mind:

1. If the SPL is subject to a single homogeneous software license, which may often be the case when a single vendor or government contractor has developed the SPL, then the license may act to reinforce a vendor lock-in situation with its customers. One of the motivating factors for OA is the desire to avoid such lock-in, whether or not the SPL components have open or standards-compliant APIs. However, a single license simplifies determination of the software ecosystem in which these system is located.

2. If an OA system employs a reference architecture, then such a reference or design-time architecture effectively defines an SPL consisting of possible different system instantiations composed from similar components from different producers (e.g., different but equivalent Web browsers, word processors, calendaring and email applications). This can be seen in the design-time architecture depicted in Figure 5, the build-time architecture in Figure 6, and the instantiated run-time architectures in Figures 7, 8, and 9.
3. If the SPL is based on an OA that integrates software components from multiple producers or OSS components that are subject to different heterogeneous licenses, then we have the situation analogous to what we have presented in this paper, but now in the form of virtual SPLs from a virtual software production enterprise [26]. that spans multiple independent OSS projects and software production enterprises. As such, SPL concepts are compatible with OA systems that are composed from heterogeneously licensed components, but do not impact the formation or evolution of the software ecosystem niches where such systems may reside.

Our approach for using open software system architectures and component licenses as a lens that focuses attention to certain kinds of relationships within and across software supply networks, software ecosystems, and networks of software ecosystems has yet to be applied to systems on mobile computing platforms. Bosch [7] notes this is a neglected area of study, but one that may offer interesting opportunities for research and software product development. Thus, what happens when we consider Apple iPhone/iPad OS, Google Android OS phones, Nokia Symbian OS phones, Nokia Maemo OS hand-held computers, Microsoft Windows 7 OS phones, Intel Moblin OS netbooks, or Nintendo DS portable game consoles as possible platforms for OA system design and deployment? First, all of these devices are just personal computers with operating systems, albeit in small, easy to carry, and wireless form factors. They represent a mix of mostly proprietary operating system platforms, though some employ

Linux-based or other OSS alternative operating systems. Second, Mobile OS platforms owners (Apple, Nokia, Google, Microsoft) are all acting to control the software ecosystems for consumers of their devices through establishment of logically centralized (but possibly physically decentralized) application distribution repositories or online stores, where the mobile device must invoke a networked link to the repository to acquire (for fee/free) and install apps. Apple has had the greatest success in this strategy and dominates the global mobile application market and mobile computing software ecosystems. But overall, OA systems are not necessarily excluded from these markets or consumers. Third, given our design-time architecture of the case study system shown in Figure 5, is it possible to identify a build-time version that could produce a run-time version that could be deployed on most or all of these mobile devices? One such build-time architecture would compose an Opera Web browser, with Web services for word processing, calendaring and email, that could be hosted on either proprietary or OSS mobile operating systems. This alternative arises since Opera Software has produced run-time versions of its proprietary Web browser for these mobile operating systems, for accessing the Web via a wireless/cellular phone network connection. Similarly, in Figure 13 the instance architecture on the right could evolve to operate on a mobile platform like an Android-based mobile device or Symbian-based cell phone. So it appears that mobile computing devices do not pose any unusual challenges for our approach in terms of understanding their software ecosystems or the ecosystem niches for OA systems that could be hosted on such devices.

7. Conclusion

The role of software ecosystems in the development and evolution of heterogeneously-licensed open architecture systems has received insufficient consideration. Such systems are composed of components potentially under two or more licenses, open source software or proprietary or both, in an architecture in which evolution can occur by evolving existing components, replacing them, or refactoring. The software licenses of the components both facilitate and constrain

in which ecosystems a composed system may lie. In addition, the obligations and rights carried by the licenses are transmitted from the software component producers to system consumers through the architectural choices made by system integrators. Thus software component licenses help determine the contours of the software supply network and software ecosystem niche that emerge for a given implementation of a composed system architecture. Accordingly, we described examples for systems whose host software platform span the range of personal computer operating systems, Web services, and mobile computing devices.

Consequently, software component licenses and the architectural composition of a system determine the software ecosystem niche where a systems lies. Understanding and describing software ecosystem niches is a key contribution of this work. A case study of an open architecture software system that articulates different niches was employed to this end. We examined how the architecture and software component licenses of a composed system at design-time, build-time, and run-time helps determine the system’s software ecosystem niche, and provides insight for identifying potential evolutionary paths of software system, architecture, and niches. Similarly, we detailed the ways in which a composed system can evolve over time, and how a software system’s evolution can change or shift the software ecosystem niche in which the system resides and thus producer-consumers relationships. Then we described how virtual software product lines can be identified through a lens that examines the association between open architectures, software component licenses, and software ecosystems.

Finally, in related work [2, 3, 4] we identified structures for modeling software licenses and the license architecture of a system, and automated support for calculating its rights and obligations. Such capabilities are needed in order to manage and track an OA system’s evolution in the context of its ecosystem niche. We have outlined an approach for achieving these structures and support and sketched how they further the goal of reusing and exchanging alternative software components and software architectural compositions. More work re-

mains to be done, but we believe this approach transforms a vexing problem of stating in detail how study of software ecosystems can be tied to core issues in software engineering like software architecture, product lines, component-based reuse, license management, and evolution, into a manageable one for which workable solutions can be obtained.

Acknowledgments

This research is supported by grants #0534771 and #0808783 from the U.S. National Science Foundation, and the Acquisition Research Program at the Naval Postgraduate School. No review, approval, or endorsement is implied.

References

- [1] Alspaugh, T. A., Antón, A. I., 2008. Scenario support for effective requirements. *Information and Software Technology* 50 (3), 198–220.
- [2] Alspaugh, T. A., Asuncion, H. U., Scacchi, W., May 2009. Analyzing software licenses in open architecture software systems. In: 2nd International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS). pp. 1–4.
- [3] Alspaugh, T. A., Asuncion, H. U., Scacchi, W., 2009. Intellectual property rights requirements for heterogeneously-licensed systems. In: 17th IEEE International Requirements Engineering Conference (RE'09). pp. 24–33.
- [4] Alspaugh, T. A., Asuncion, H. U., Scacchi, W., 2009. The role of software licenses in open architecture ecosystems. In: First International Workshop on Software Ecosystems (IWSECO-2009). pp. 4–18.
- [5] Bass, L., Clements, P., Kazman, R., 2003. *Software Architecture in Practice*. Addison-Wesley Longman.
- [6] Bosch, J., 2000. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley.

- [7] Bosch, J., 2009. From software product lines to software ecosystems. In: 13th International Software Product Line Conference (SPLC'09). pp. 111–119.
- [8] Bosch, J., Bosch-Sijtsema, P., 2010. From integration to composition: On the impact of software product lines, global development and ecosystems. *J. Syst. Softw.* 83 (1), 67–76.
- [9] Boucharas, V., Jansen, S., Brinkkemper, S., 2009. Formalizing software ecosystem modeling. In: First International Workshop on Open Component Ecosystems (IWOCE'09). pp. 41–50.
- [10] Breaux, T. D., Anton, A. I., 2005. Analyzing goal semantics for rights, permissions, and obligations. In: RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering. pp. 177–188.
- [11] Breaux, T. D., Anton, A. I., 2008. Analyzing regulatory rules for privacy and security requirements. *IEEE Transactions on Software Engineering* 34 (1), 5–20.
- [12] Brown, A. W., Booch, G., 2002. Reusing open-source software and practices: The impact of open-source on commercial vendors. In: *Software Reuse: Methods, Techniques, and Tools (ICSR-7)*. pp. 381–428.
- [13] Clements, P., Northrop, L., 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional.
- [14] Dashofy, E. M., 2007. Supporting stakeholder-driven, multi-view software architecture modeling. Ph.D. thesis, University of California, Irvine.
- [15] Feldt, K., 2007. *Programming Firefox: Building Rich Internet Applications with XUL*. O'Reilly Media, Inc.
- [16] Firesmith, D., Jan.–Feb. 2004. Specifying reusable security requirements. *Journal of Object Technology* 3 (1), 61–75.

- [17] German, D. M., Hassan, A. E., 2009. License integration patterns: Dealing with licenses mismatches in component-based development. In: 28th International Conference on Software Engineering (ICSE '09). pp. 188–198.
- [18] Jansen, S., Brinkkemper, S., Finkelstein, A., 2009. Business network management as a survival strategy: A tale of two software ecosystems. In: First Workshop on Software Ecosystems. pp. 34–48.
- [19] Jansen, S., Finkelstein, A., Brinkkemper, S., 2009. A sense of community: A research agenda for software ecosystems. In: 28th International Conference on Software Engineering (ICSE '09), Companion Volume. pp. 187–190.
- [20] Jensen, C., Scacchi, W., Jul./Sep. 2005. Process modeling across the web information infrastructure. *Software Process: Improvement and Practice* 10 (3), 255–272.
- [21] Kuehnelt, A.-K., Jun. 2008. Microsoft, open source and the software ecosystem: of predators and prey—the leopard can change its spots. *Information & Communication Technology Law* 17 (2), 107–124.
- [22] Kuhl, F., Weatherly, R., Dahmann, J., 1999. *Creating computer simulation systems: an introduction to the high level architecture*. Prentice Hall.
- [23] Messerschmitt, D. G., Szyperski, C., 2003. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. MIT Press.
- [24] Meyers, B. C., Oberndorf, P., 2001. *Managing Software Acquisition: Open Systems and COTS Products*. Addison-Wesley Professional.
- [25] Nelson, L., Churchill, E. F., 2006. Repurposing: Techniques for reuse and integration of interactive systems. In: *International Conference on Information Reuse and Integration (IRI-08)*. p. 490.
- [26] Noll, J., Scacchi, W., Feb. 1999. Supporting software development in virtual enterprises. *J. of Digital Information* 1 (4).

- [27] Noll, J., Scacchi, W., 2001. Specifying process-oriented hypertext for organizational computing. *J. Network and Computing Applications* 24 (1), 39–61.
- [28] Open Source Initiative, 2010. Open Source Definition. <http://www.opensource.org/docs/osd>.
- [29] Oreizy, P., 2000. Open architecture software: A flexible approach to decentralized software evolution. Ph.D. thesis, University of California, Irvine.
- [30] Ovaska, P., Rossi, M., Marttiin, P., 2003. Architecture as a coordination tool in multi-site software development. *Software Process: Improvement and Practice* 8 (4), 233–247.
- [31] Rosen, L., 2005. Open Source Licensing: Software Freedom and Intellectual Property Law. Prentice Hall.
- [32] Scacchi, W., 2007. Free/open source software development: Recent research results and emerging opportunities. In: 6th Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007). pp. 459–468.
- [33] Scacchi, W., Alspaugh, T. A., May 2008. Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In: 5th Annual Acquisition Research Symposium.
- [34] Taylor, R. N., Medvidovic, N., Dashofy, E. M., 2009. Software Architecture: Foundations, Theory, and Practice. Wiley.
- [35] Unity Technologies, Dec. 2008. End User License Agreement. <http://unity3d.com/unity/unity-end-user-license-2.x.html>.
- [36] van Gurp, J., Prehofer, C., Bosch, J., 2010. Comparing practices for reuse in integration-oriented software product lines and large open source software projects. *Software — Practice & Experience* 40 (4), 285–312.

- [37] Ven, K., Mannaert, H., 2008. Challenges and strategies in the use of open source software by independent software vendors. *Information and Software Technology* 50 (9-10), 991–1002.
- [38] Yau, S. S., Chen, Z., 2006. A framework for specifying and managing security requirements in collaborative systems. In: *Third International Conference on Autonomic and Trusted Computing (ATC 2006)*. pp. 500–510.

Industry Data on Young Users in Virtual Worlds Worldwide

Main Source: <http://www.kzero.co.uk/index.php> and others as indicated, circa Feb-Apr 2010.

Note: Not an endorsement nor claim for validity of this data

Note: figures below are expressed in units of millions of users worldwide

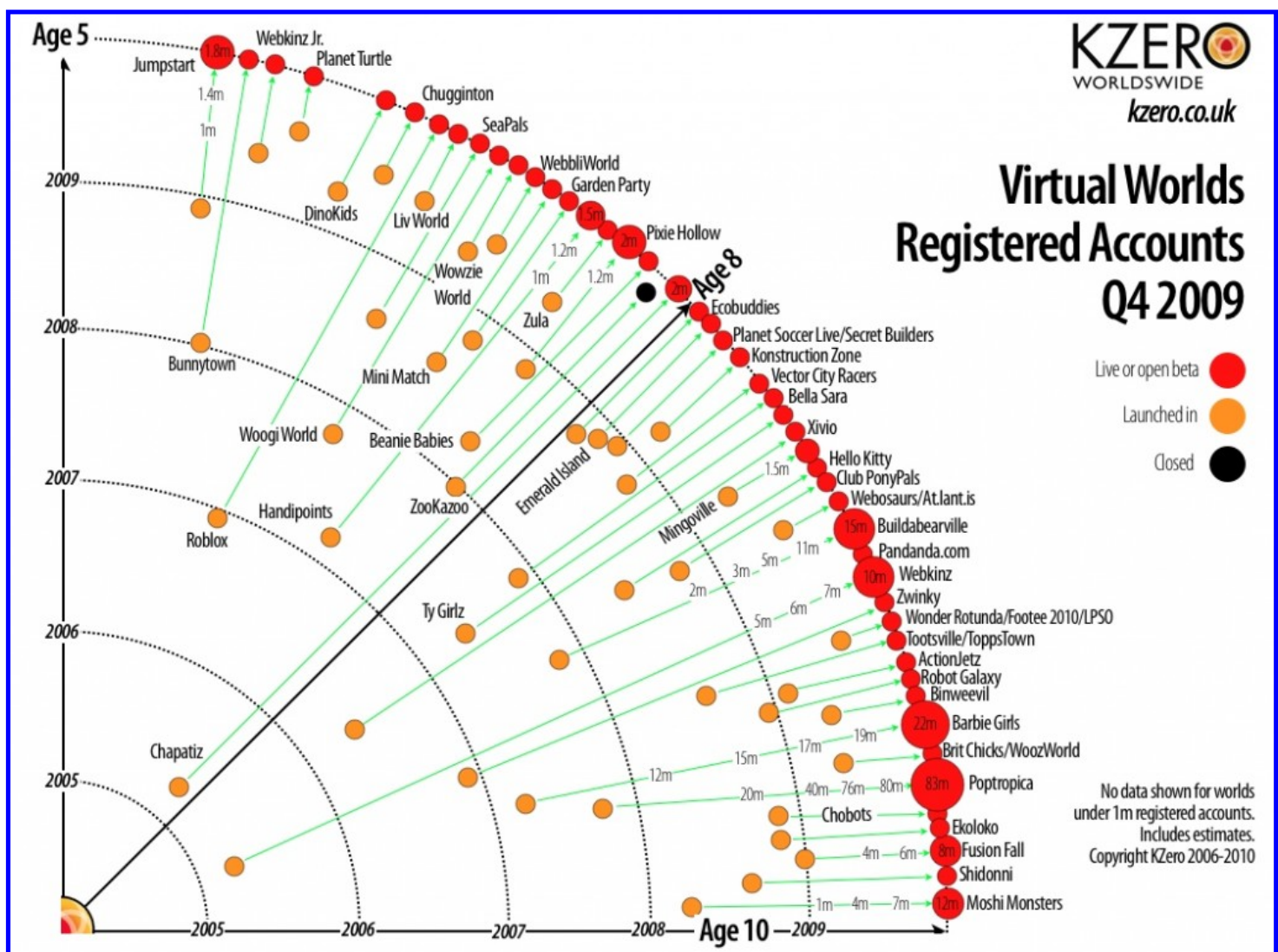
Note: click on the images for larger/detailed view

Note: copyrighted material from KZero and others, displayed here for research discussion purposes only. do not repost.

Q4 2009 Universe chart: Kids and Tweens

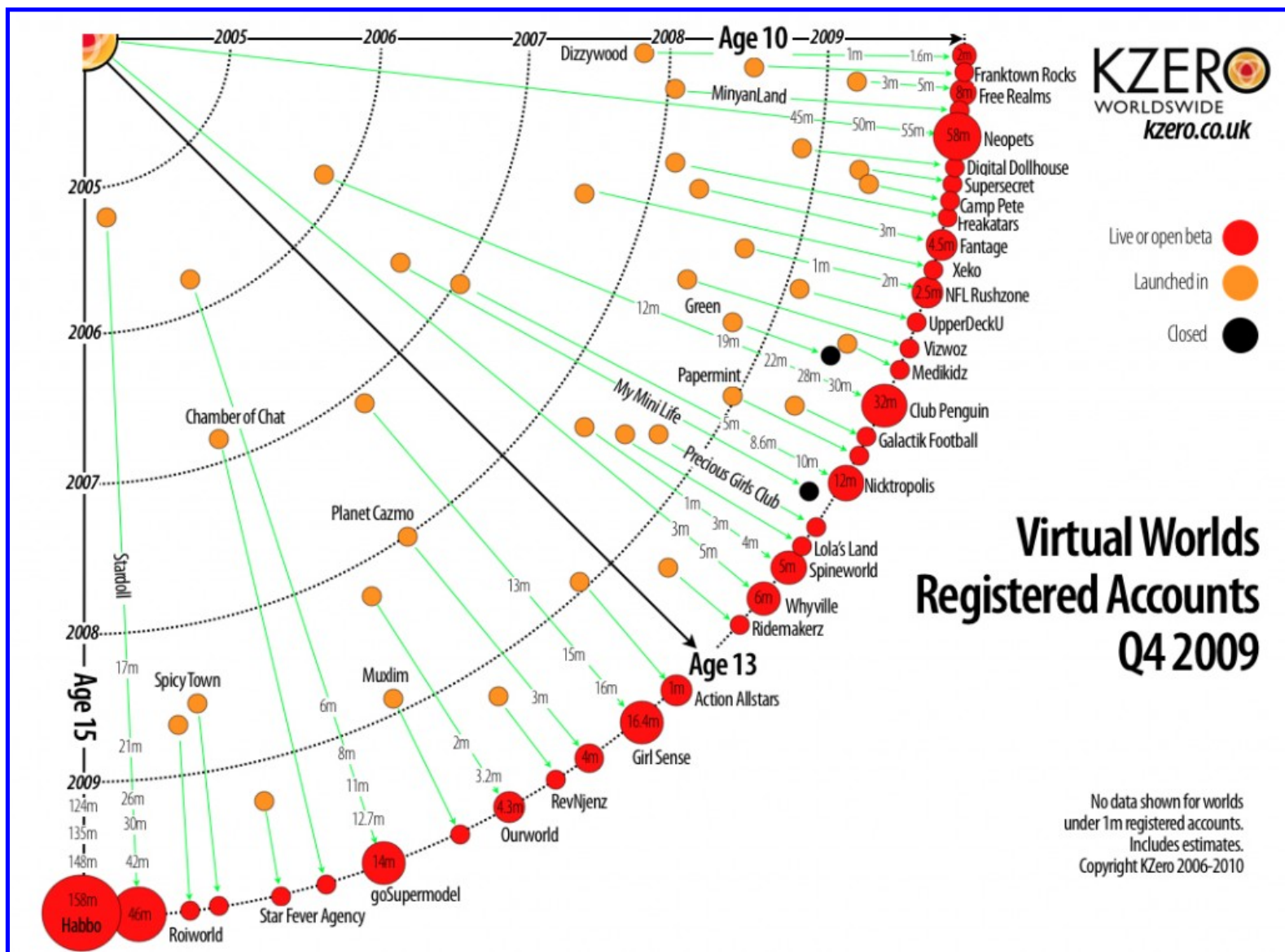
As published in their [last post](#), the overall market of total registered users in the virtual worlds sector reached [800m in Q4 2009](#). Here, we delve into the younger segmented of this market, Kids and Tweens.

Virtual worlds with an average age user between five and ten reached a total of 179m in Q4, up 17.8% from 152m. The chart below contains the Universe segment for this age range.



Commentary on this age range in terms of drivers for growth and upcoming worlds is explained in this post based on the Radar. The [full report on Q4](#) growth containing both the Universe and Radar charts can be requested [here](#).

Here's the segment for the ten to 15 year old group. This segment (the largest in overall size) grew 6.8% quarter on quarter from 367m to 392m.

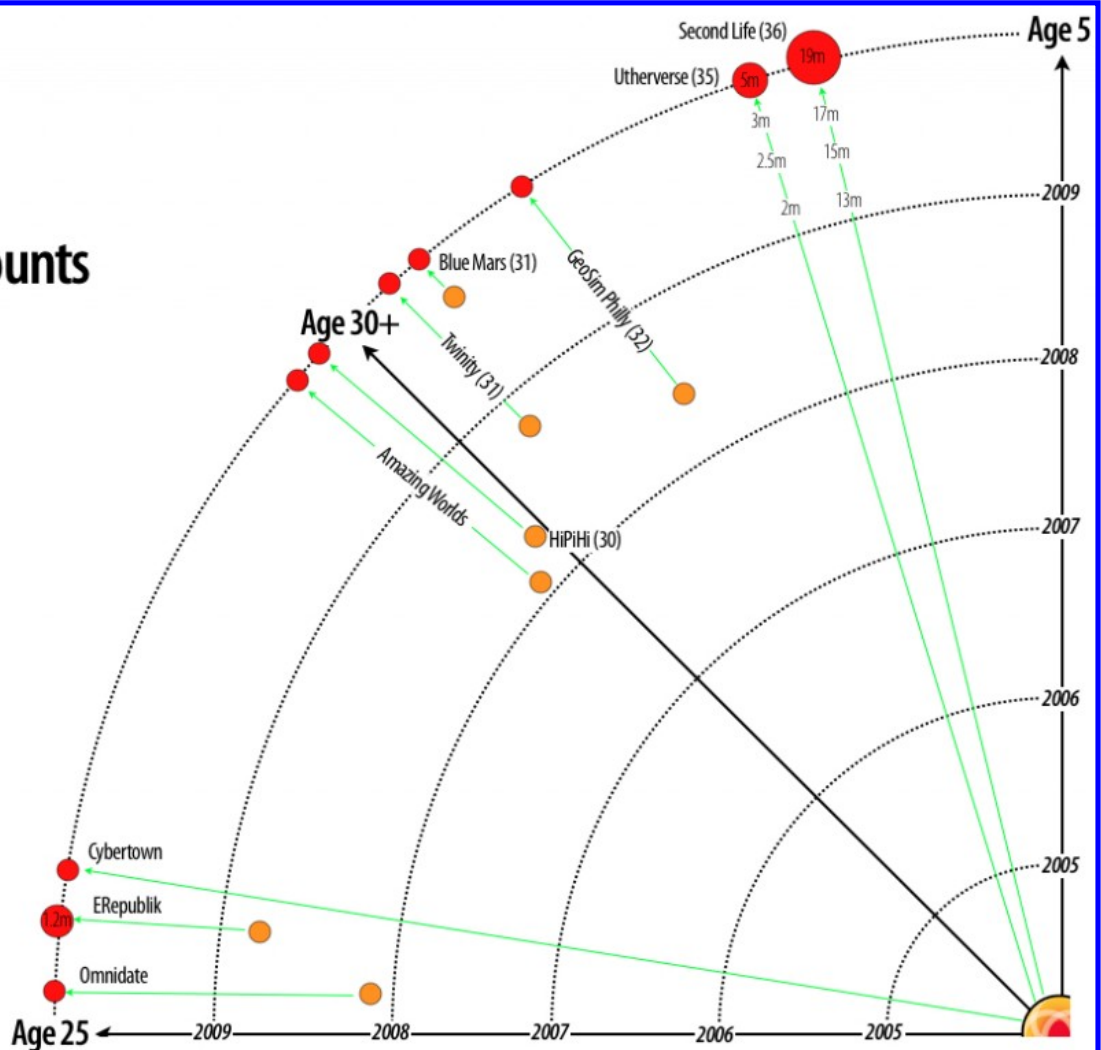


Our last post (assessing [Kids and Tweens](#)) showed the five to 10 year old segment and 10 to 15's grew 17.8% and 6.8% respectively from Q3 to Q4 2009. Here, we cover the older sectors and first up virtual worlds with an average age user from 15 to 25. Here's the Universe chart.

Virtual Worlds Registered Accounts Q4 2009

- Live or open beta
- Launched in
- Closed

No data shown for worlds under 1m registered accounts.
Includes estimates.
Copyright KZero 2006-2010

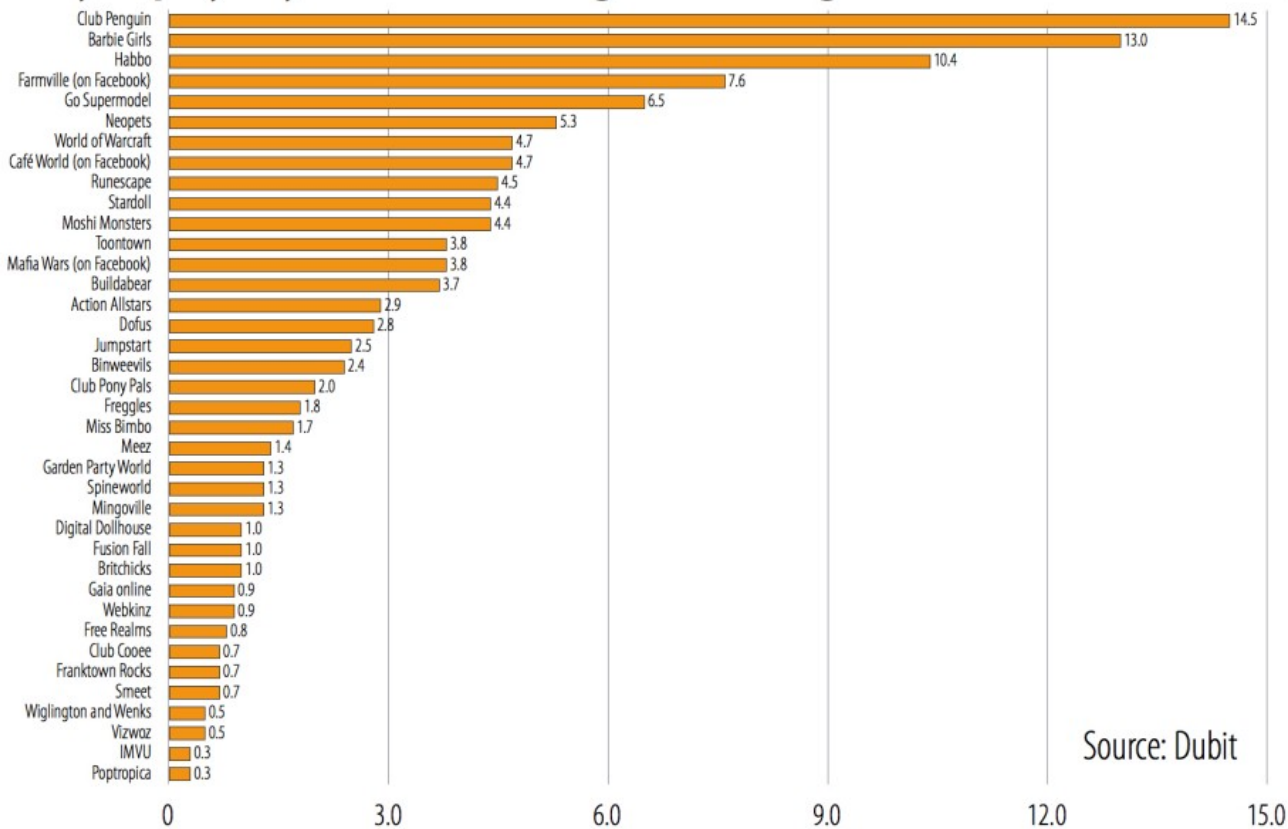


Kids virtual world popularity across the EU -- <http://www.kzero.co.uk/blog/?p=4111>

Here's the master summary of the [Dubit](#) research looking at kids virtual worlds. The countries included in this research were: UK, France, Germany, Holland, Sweden, Finland, Norway and Denmark.

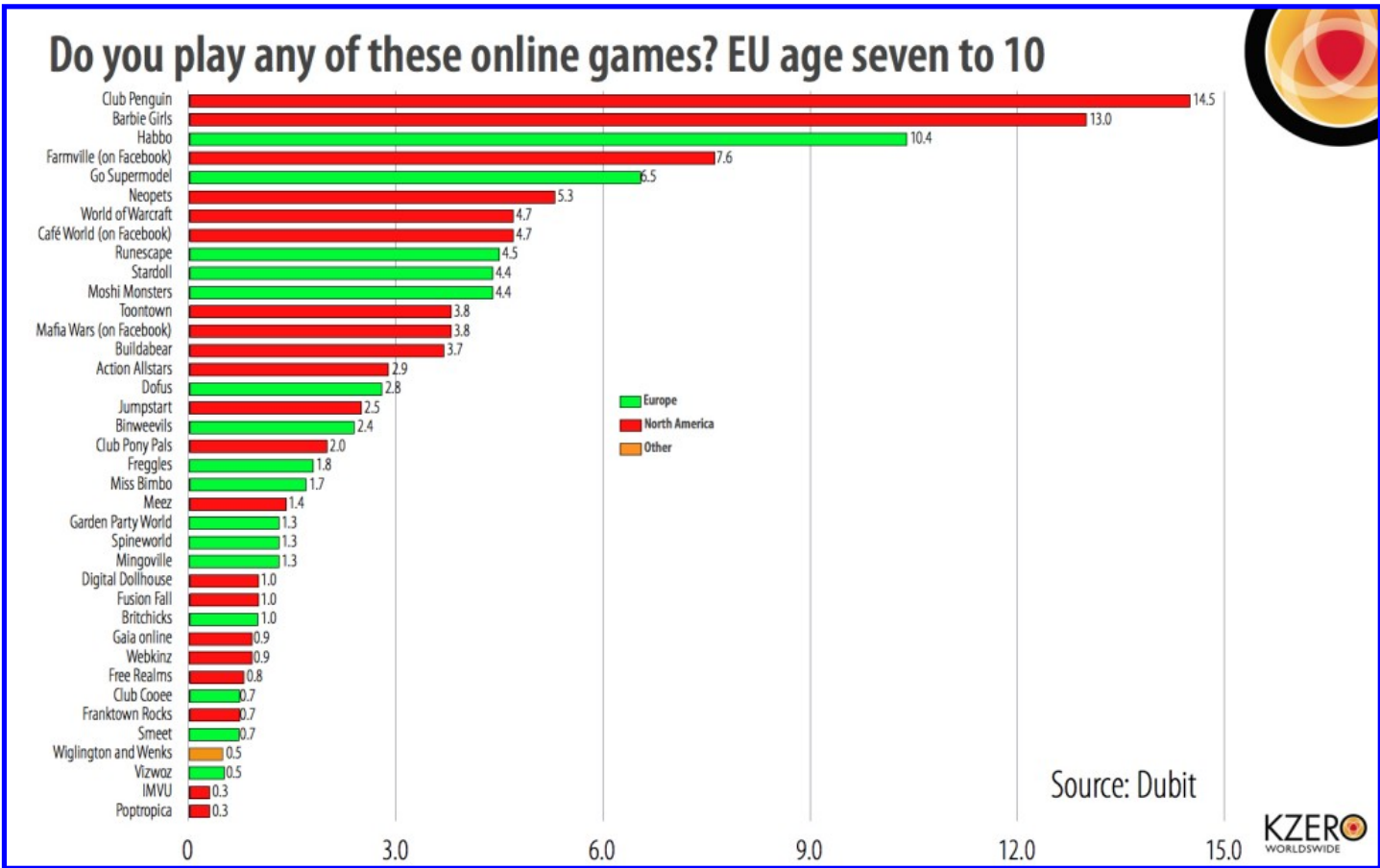
Below is the summary slide ranked by the % of the sample that has played/registered each world.

Do you play any of these online games? EU age seven to 10



Club Penguin comes out top with 14.5% of the sample having played it. Second place goes to Barbie Girls with 13% with Habbo in third. Interestingly GoSupermodel beats Stardoll based on this research, albeit by a couple of % points.

Looking at this EU summary from a company-location perspective...



Excluding Facebook Games and older MMOs, Habbo takes the crown at the top European-based VW, followed by GoSupermodel. Moshi Monsters and Stardoll are next in popularity.

Virtual world registered accounts reach 800m

At the **end of Q4 2009**, total registered accounts in the virtual worlds sector reached 803m. This is a 19.7% (132m) quarter on quarter increase, from 671m registered accounts in Q3. The table below breaks this out by average user age range.

Age Range	Q1	Q2	Q3	Q4
5 to 10	77m	114m	152m	179m
10 to 15	246m	334m	367m	392m
15 to 25	73m	99m	117m	193m
25+	23m	32m	35m	39m
Total	419m	579m	671m	803m

Over the last four quarters on a registered accounts basis the market has almost doubled, going from 419m to 803m.

Looking at this growth by age range (the average user age), the 15 to 25 year old segment demonstrated the highest growth in Q4, representing a 65% increase from 117m to 193m. Strong performance from [IMVU](#) and [Habbo](#) drove this uplift.

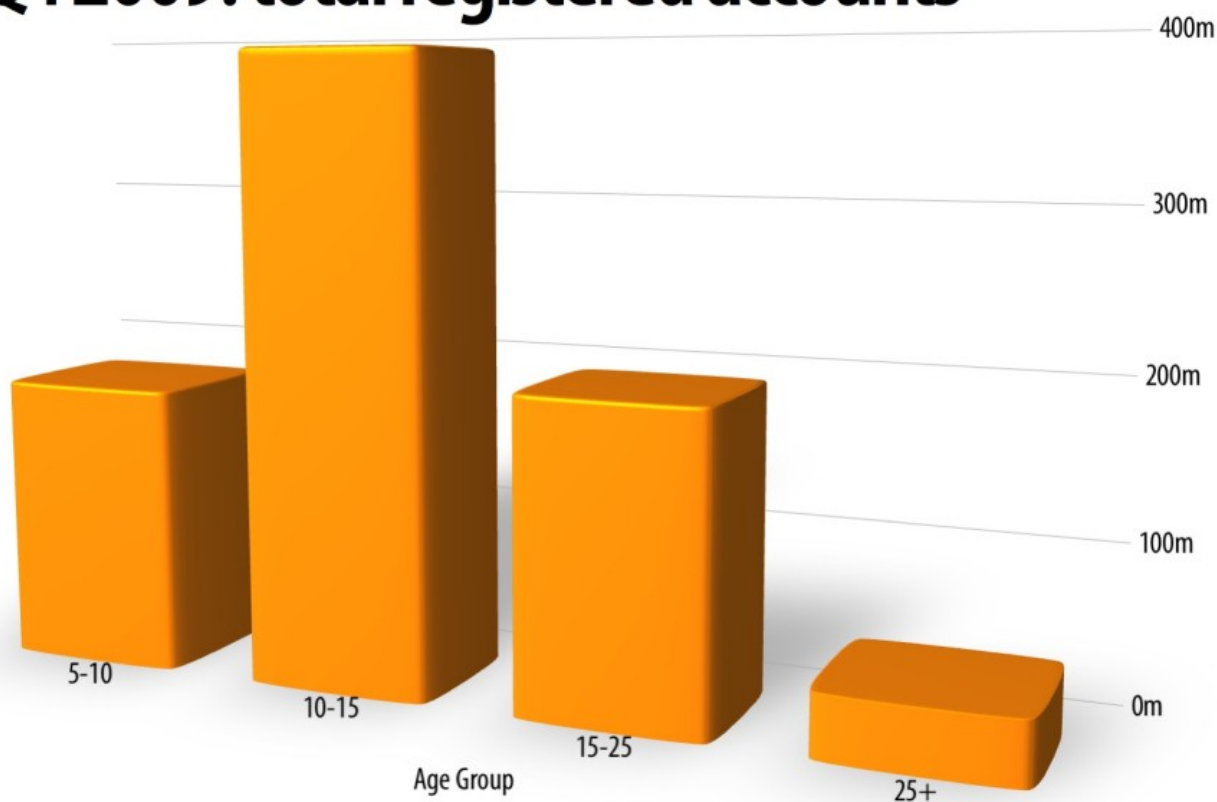
Second highest growth came in the youngest age range (five to 10 year old) and as the just released Radar chart shows, this segment, in particular for virtual worlds catering to education and development is hotting up. Quarter on quarter growth in this segment was 27m, moving from 152m to 179m.

Our Universe and Radar charts have also been updated based on closing Q4 data. Here's a post for the [Universe](#) and another for the [Radar](#).

The full report covering growth in the virtual worlds sector can be ordered [here](#).

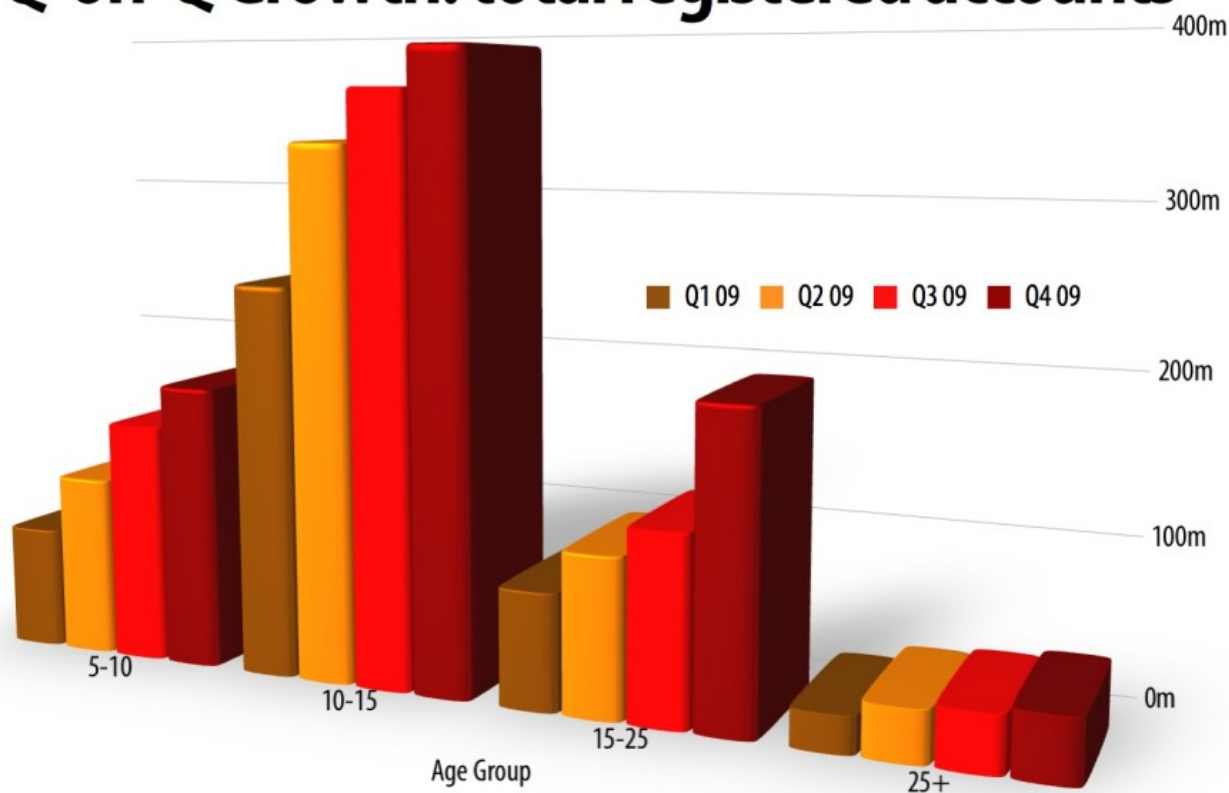
Here's (of course) a few of charts.

Q4 2009: total registered accounts



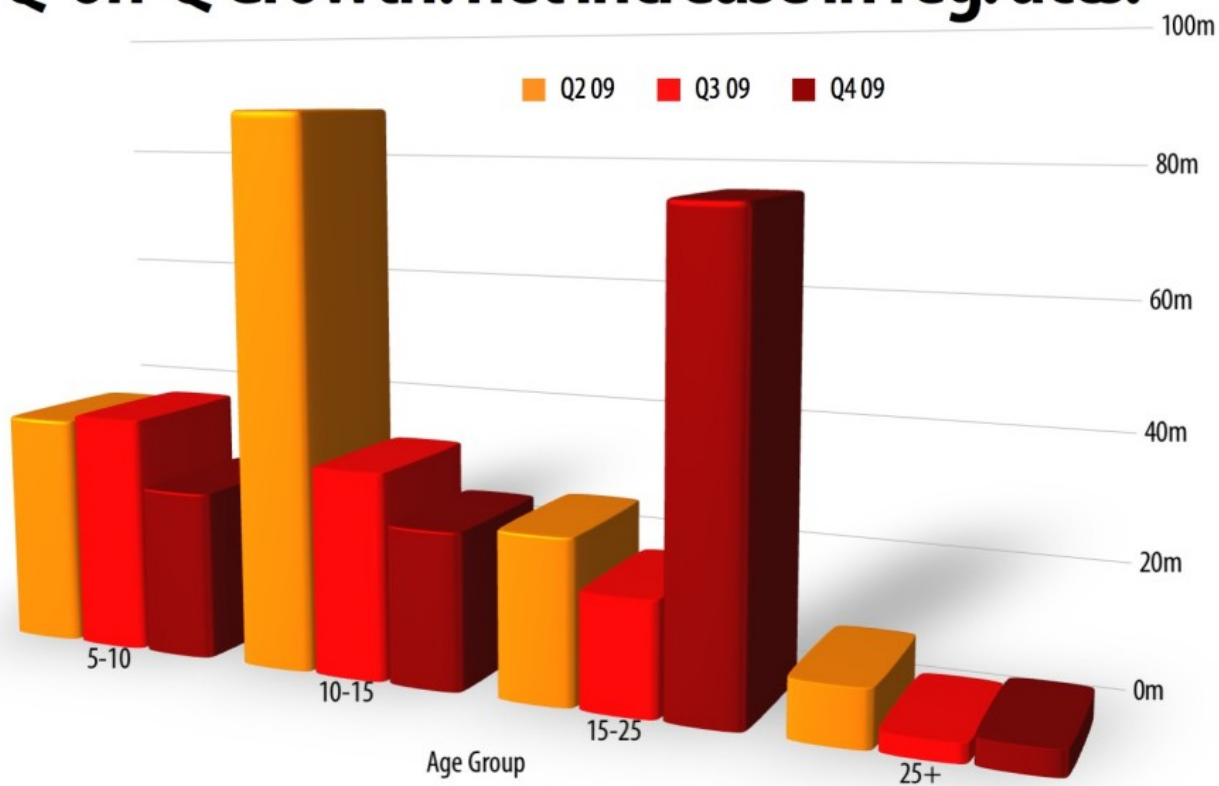
KZERO
WORLDWIDE

Q-on-Q Growth: total registered accounts



KZERO
WORLDWIDE

Q-on-Q Growth: net increase in reg. accs.



Software Licenses, Open Source Components, and Open Architectures

Thomas A. Alspaugh, Hazeline U. Asuncion, and Walt Scacchi

Institute for Software Research

University of California, Irvine

Irvine, CA 92697-3455 USA

{alspaugh,hasuncion,wcacchi}@ics.uci.edu

Abstract

A substantial number of enterprises and independent software vendors are adopting a strategy in which software-intensive systems are developed with an open architecture (OA) that may contain open source software (OSS) components or components with open APIs. The emerging challenge is to realize the benefits of openness when components are subject to different copyright or property licenses. In this paper we identify key properties of OSS licenses, present a license analysis scheme to identify license conflicts arising from composed software elements, and apply it to provide guidance for software architectural design choices whose goal is to enable specific licensed component configurations. Our scheme has been implemented in an operational environment and demonstrates a practical, automated solution to the problem of determining overall rights and obligations for alternative OAs.

1. Introduction

It has been common for OSS projects to require that developers contribute their work under conditions that ensure the project can license its products under a specific OSS license. For example, the Apache Contributor License Agreement grants enough rights to the Apache Software Foundation for the foundation to license the resulting systems under the Apache License. This sort of license configuration, in which the rights to a system's components are homogenously granted and the system has a well-defined OSS license, was the norm and continues to this day.

However, we more and more commonly see a different license configuration, in which the components of a system do not have the same license. The resulting system may not have any recognized OSS license at all—in fact, our research indicates this is the most likely

outcome—but instead, if all goes well in its design, there will be enough rights available in the system so that it can be used and distributed, and perhaps modified by others and sublicensed, if the corresponding obligations are met. These obligations are likely to differ for components with different licenses; a BSD (Berkeley Software Distribution) licensed component must preserve its copyright notices when made part of the system, for example, while the source code for a modified component covered by MPL (the Mozilla Public License) must be made public, and a component with a reciprocal license such as the Free Software Foundation's GPL (General Public License) might carry the obligation to distribute the source code of that component but also of other components that constitute “a whole which is a work based on” the GPL'd component. The obligations may conflict, as when a GPL'd component's reciprocal obligation to publish source code of other components is combined with a proprietary license's prohibition of publishing source code, in which case there may be no rights available for the system as a whole, not even the right of use, because the obligations of the licenses that would permit use of its components cannot simultaneously be met.

The central problem we examine and explain in this paper is to identify principles of software architecture and software licenses that facilitate or inhibit success of the OA strategy when OSS and other software components with open APIs are employed. This is the knowledge we seek to develop and deliver. Without such knowledge, it is unlikely that an OA that is clean, robust, transparent, and extensible can be readily produced. On a broader scale, this paper seeks to explore and answer the following kinds of research questions:

- What license applies to an OA system composed with components with different licenses?
- How do alternative OSS licenses facilitate or inhibit the development of OA systems?

- How should software license constraints be specified so it is possible to automatically determine the overall set of rights and obligations associated with a configured software system architecture?

This paper may help establish a foundation for how to analyze and evaluate dependencies that might arise when seeking to develop software systems that embody an OA when different types of software components or software licenses are being considered for integration into an overall system configuration.

In the remainder of this paper, we examine software licensing constraints. This is followed by an analysis of how these constraints can interact in order to determine the overall license constraints applicable to the configured system architecture. Next, we describe an operational environment that demonstrates automatic determination of license constraints associated with a configured system architecture, and thus offers a solution to the problem we face. We close with a discussion of the conclusions that follow.

2. Background

There is little explicit guidance or reliance on systematic empirical studies for how best to develop, deploy, and sustain complex software systems when different OA and OSS objectives are at hand. Instead, we find narratives that provide ample motivation and belief in the promise and potential of OA and OSS without consideration of what challenges may lie ahead in realizing OA and OSS strategies. Ven [2008] is a recent exception.

We believe that a primary challenge to be addressed is how to determine whether a system, composed of subsystems and components each with specific OSS or proprietary licenses, and integrated in the system's planned configuration, is or is not open, and what license constraints apply to the configured system as a whole. This challenge comprises not only evaluating an existing system at run-time, but also at design-time and build-time for a proposed system to ensure that the result is "open" under the desired definition, and that only the acceptable licenses apply; and also understanding which licenses are acceptable in this context. Because there are a range of types and variants of licenses [cf. OSI 2008], each of which may affect a system in different ways, and because there are a number of different kinds of OSS-related components and ways of combining them that affect the licensing issue, a first necessary step is to understand the kinds of software elements that constitute a software architecture, and what kinds of licenses may encumber these elements or their overall configuration.

OA seems to simply mean software system architectures incorporating OSS components and open application program interfaces (APIs). But not all software system architectures incorporating OSS components and open APIs will produce an OA, since the openness of an OA depends on: (a) how/why OSS and open APIs are located within the system architecture, (b) how OSS and open APIs are implemented, embedded, or interconnected, (c) whether the copyright (Intellectual Property) licenses assigned to different OSS components encumber all/part of a software system's architecture into which they are integrated, and (d) the fact that many alternative architectural configurations and APIs exist that may or may not produce an OA [cf. Antón and Alspaugh 2007, Scacchi and Alspaugh 2008]. Subsequently, we believe this can lead to situations in which new software development or acquisition requirements stipulate a software system with an OA and OSS, but the resulting software system may or may not embody an OA. This can occur when the architectural design of a system constrains system requirements—raising the question of what requirements can be satisfied by a given system architecture, when requirements stipulate specific types or instances of OSS (e.g., Web browsers, content management servers) to be employed, or what architecture style [Bass, Clements, and Kazman 2003] is implied by a given set of system requirements.

Thus, given the goal of realizing an OA and OSS strategy together with the use of OSS components and open APIs, it is unclear how to best align acquisition, system requirements, software architectures, and OSS elements across different software license regimes to achieve this goal [Scacchi and Alspaugh 2008].

3. Understanding open architectures

The statement that a system is intended to embody an open architecture using open software technologies like OSS and APIs, does not clearly indicate what possible mix of software elements may be configured into such a system. To help explain this, we first identify what kinds of software elements are included in common software architectures whether they are open or closed [cf. Bass, Clements, Kazman 2003].

1. Software source code components – (a) standalone programs, (b) libraries, frameworks, or middleware, (c) inter-application script code (e.g., C shell scripts) and (d) intra-application script code (e.g., to create Rich Internet Applications using domain-specific languages (e.g., XUL for Firefox Web

browser [Feldt 2007] or “mashups” [Nelson and Churchill 2006]).

2. Executable components -- These are programs for which the software is in binary form, and its source code may not be open for access, review, modification, and possible redistribution. Executable binaries can be viewed as “derived works” [Rosen 2005].
3. Application program interfaces/APIs – The availability of externally visible and accessible APIs to which independently developed components can be connected is the minimum condition required to form an “open system” [Meyers and Obendorf 2001].
4. Software connectors – In addition to APIs, these may be software either from libraries, frameworks, or application script code whose intended purpose is to provide a standard or reusable way of associating programs, data repositories, or remote services through common interfaces. The High Level Architecture (HLA) is an example of a software connector scheme [Kuhl, Weatherly, Damann 2000], as are CORBA, Microsoft’s .NET, Enterprise Java Beans, and LGPL libraries.
5. Configured system or sub-system architectures – These are software systems that can be built to conform to an explicit architectural design. They include software source code components, executable components, APIs, and connectors that are organized in a way that may conform to a known “architectural style” such as the Representational State Transfer [Fielding and Taylor 2002] for Web-based client-server applications, or may represent an original or ad hoc architectural pattern [Bass 2003]. Each of the software elements, and the pattern in which they are arranged and inter-linked, can all be specified, analyzed, and documented using an Architecture Description Language and ADL-based support tools [Bass 2003, Medvidovic 1999].

Figure 1 provides an overall view of an archetypal software architecture for a configured system that includes and identifies each of the software elements above, as well as including free/open source software (e.g., Gnome Evolution) and closed source software (WordPerfect) components. In simple terms, the configured system consists of software components (grey boxes in the Figure) that include a Mozilla Web browser, Gnome Evolution email client, and WordPerfect word processor, all running on a Linux operating system that can access file, print, and other remote networked servers (e.g. an Apache Web server). These

components are interrelated through a set of software connectors (ellipses in the Figure) that connect the interfaces of software components (small white boxes attached to a component) and link them together. Modern day enterprise systems or command and control systems will generally have more complex architectures and a more diverse mix of software components than shown in the figure here. As we examine next, even this simple architecture raises a number of OSS licensing issues that constrain the extent of openness that may be realized in a configured OA.

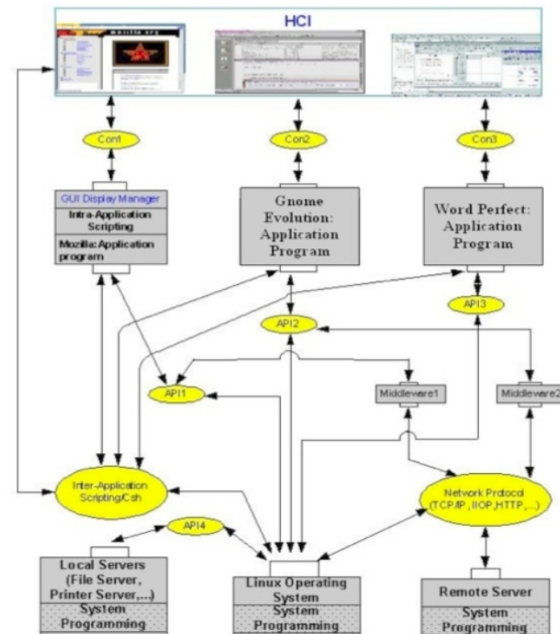


Figure 1. An archetypal software architecture depicting components (grey boxes), connectors (ellipses), interfaces (small boxes on components), and data/control links

4. Understanding open software licenses

A particularly knotty challenge is the problem of licenses in OSS and OA. There are a number of different OSS licenses, and their number continues to grow. Each license stipulates different constraints attached to software components that bear it. External references are available which describe and explain many different licenses that are now in use with OSS [Fontana 2008, OSI 2008, Rosen 2005, St. Laurent 2004].

More and more software systems are designed, built, released, and distributed as OAs composed of components from different sources, some proprietary and others not. Systems include components that are statically bound or interconnected at build-time, while other

components may only be dynamically linked for execution at run-time, and thus might not be included as part of a software release or distribution. Software components in such systems evolve not only by ongoing maintenance, but also by architectural refactoring, alternative component interconnections, and component replacement (via maintenance patches, installation of new versions, or migration to new technologies). Software components in such systems may be subject to different software licenses, and later versions of a component may be subject to different licenses (e.g., from CDDL (Sun's Common Development and Distribution License) to GPL, or from GPLv2 to GPLv3).

Software systems with open architectures are subject to different software licenses than may be common with traditional proprietary, closed source systems from a single vendor. Software architects/developers must increasingly attend to how they design, develop, and deploy software systems that may be subject to multiple, possibly conflicting software licenses. We see architects, developers, software acquisition managers, and others concerned with OAs as falling into three groups. The first group pays little or no heed to license conflicts and obligations; they simply focus on the other goals of the system. Those in the second group have assets and resources, and to protect these they may have an army of lawyers to advise them on license issues and other potential vulnerabilities; or they may constrain the design of their systems so that only a small number of software licenses (possibly just one) are involved, excluding components with other licenses independent of whether such components represent a more effective or more efficient solution. The third group falls between these two extremes; members of this group want to design, develop, and distribute the best systems possible, while respecting the constraints associated with different software component licenses. Their goal is a configured OA system that meets all its goals, and for which all the license obligations for the needed copyright rights are satisfied. It is this third group that needs the guidance the present work seeks to provide.

There has been an explosion in the number, type, and variants of software licenses, especially with open source software (cf. OSI 2008). Software components are now available subject to licenses such as the General Public License (GPL), Mozilla Public License (MPL), Apache Public License, (APL), Academic licenses (e.g., BSD, MIT), Creative Commons, Artistic, and others as well as Public Domain (either via explicit declaration or by expiration of prior copyright license). Furthermore, licenses such as these can evolve, resulting in new license versions over time. But no matter

their diversity, software licenses represent a legally enforceable contract that is recognized by government agencies, corporate enterprises, individuals, and judicial courts, and thus they cannot be taken trivially. As a consequence, software licenses constrain open architectures, and thus architectural design decisions.

So how might we support the diverse needs of different software developers, with respect to their need to design, develop, and deploy configured software systems with different, possibly conflicting licenses for the software components they employ? Is it possible to provide automated means for helping software developers determine what constraints will result at design-time, build-time, or run-time when their configured system architectures employ diverse licensed components? These are the kind of questions we address in this paper.

4.1. Software licenses: Rights and obligations

Copyright, the common basis for software licenses, gives the original author of a work certain exclusive rights, which for software include the right to use, copy, modify, merge, publication, distribution, sub-licensing, and sell copies. These rights may be licensed to others; the rights may be licensed individually or in groups, and either exclusively so that no one else can exercise them or (more commonly) non-exclusively. After a period of years, the rights enter the public domain, but until then the only way for anyone other than the author to have any of the copyright rights is to license them.

Licenses may impose obligations that must be met in order for the licensee to realize the assigned rights. Commonly cited obligations include the obligation to buy a legal copy to use and not distribute copies (proprietary licenses); the obligation to preserve copyright and license notices (academic licenses); the obligation to publish at no cost source code you modify (MPL); or the reciprocal obligation to publish all source code included at build-time or statically linked (GPL).

Licenses may provide for the creation of derivative works (e.g., a transformation or adaptation of existing software) or collective works (e.g., a Linux distribution that combines software from many independent sources) from the original work, by granting those rights possibly with corresponding obligations.

In addition, the author of an original work can make it available under more than one license, enabling the work's distribution to different audiences with different needs. For example, one licensee might be happy to pay a license fee in order to be able to distribute the work as part of a proprietary product whose source

code is not published, while another might need to license the work under MPL rather than GPL in order to have consistent licensing across a system. Thus we now see software distributed under any one of several licenses, with the licensee choosing from two (“dual license”) or three (Mozilla’s “tri-license”) licenses.

The basic relationship between software license rights and obligations can be summarized as follows: if you meet the specified obligations, then you get the specified rights. So, informally, for the academic licenses, if you retain the copyright notice, list of license conditions, and disclaimer, then you can use, modify, merge, sub-license, etc. For MPL, if you publish modified source code and sub-licensed derived works under MPL, then you get all the MPL rights. And so forth for other licenses. However, one thing we have learned from our efforts to carefully analyze and lay out the obligations and rights pertaining to each license is that license details are difficult to comprehend and track—it is easy to get confused or make mistakes. Some of the OSS licenses were written by developers, and often these turn out to be incomplete and legally ambiguous; others, usually more recent, were written by lawyers, and are more exact and complete but can be difficult for non-lawyers to grasp. The challenge is multiplied when dealing with configured system architectures that compose multiple components with heterogeneous licenses, so that the need for legal interpretations begins to seem inevitable [cf. Fontana 2008, Rosen 2005]. Therefore, one of our goals is to make it possible to architect software systems of heterogeneously-licensed components without necessarily consulting legal counsel. Similarly, such a goal is best realized with automated support that can help architects understand design choices across components with different licenses, and that can provide support for testing build-time releases and run-time distributions to make sure they achieve the specified rights by satisfying the corresponding obligations.

4.2. Expressing software licenses

Historically, most software systems, including OSS systems, were entirely under a single software license. However, we now see more and more software systems being proposed, built, or distributed with components that are under various licenses. Such systems may no longer be covered by a single license, unless such a licensing constraint is stipulated at design-time, and enforced at build-time and run-time. But when components with different licenses are to be included at build-time, their respective licenses might either be consistent or conflict. Further, if designed systems include

components with conflicting licenses, then one or more of the conflicting components must be excluded in the build-time release or must be abstracted behind an open API or middleware, with users required to download and install to enable the intended operation. (This is common in Linux distributions subject to GPL, where for example users may choose to acquire and install proprietary run-time components, like proprietary media players). So a component license conflict need not be a show-stopper if identified at design time. However, developers have to be able to determine which components’ licenses conflict and to take appropriate steps at design, build, and run times, consistent with the different concerns and requirements that apply at each phase [cf. Scacchi and Alspaugh 2008].

In order to fulfill our goals, we need a scheme for expressing software licenses that is more formal and less ambiguous than natural language, and that allows us to identify conflicts arising from the various rights and obligations pertaining to two or more component’s licenses. We considered relatively complex structures (such as Hohfeld’s eight fundamental jural relations [Hohfeld 1913]) but, applying Occam’s razor, selected a simpler structure. We start with a tuple $\langle actor, operation, action, object \rangle$ for expressing a right or obligation. The *actor* is the “licensee” for all the licenses we have examined. The *operation* is one of the following: “may”, “must”, or “must not”, with “may” expressing a right and “must” and “must not” expressing obligations; following Hohfeld, the lack of a right (which would be “may not”) correlates with a duty to not exercise the right (“must not”), and whenever lack of a right seemed significant in a license we expressed it as a negative obligation with “must not”. The *action* is a verb or verb phrase describing what may, must, or must not be done, with the *object* completing the description. We specify an object separately from the action in order to minimize the set of actions. A license then may be expressed as a set of rights, with each right associated (in that license) with zero or more obligations that must be fulfilled in order to enjoy that right. Figure 2 displays the tuples and associations for two of the rights and their associated obligations for the academic BSD software license. Note that the first right is granted without corresponding obligations.

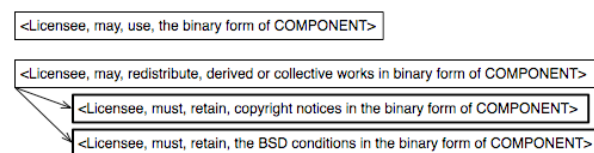


Figure 2. A portion of the BSD license tuples

We now turn to examine how OA software systems that include components with different licenses can be designed and analyzed while effectively tracking their rights and obligations.

When designing an OA software system, there are heuristics that can be employed to enable architectural design choices that might otherwise be excluded due to license conflicts. First, it is possible to employ a “license firewall” which serves to limit the scope of reciprocal obligations. Rather than simply interconnecting conflicting components through static linking of components at build time, such components can be logically connected via dynamic links, client-server protocols, license shims (e.g., via LGPL connectors), or runtime plug-ins. Second, the source code of statically linked OSS components must be made public. Third, it is necessary to include appropriate notices and publish required sources when academic licenses are employed. However, even using design heuristics such as these (and there are many), keeping track of license rights and obligations across components that are interconnected in complex OAs quickly become too cumbersome. Thus, automated support needs to be provided to help overcome and manage the multi-component, multi-license complexity.

5. Automating analysis of software license rights and obligation

We find that if we start from a formal specification of a software system’s architecture, then we can associate software license attributes with the system’s components, connectors, and sub-system architectures and calculate the copyright rights and obligations for the system. Accordingly, we employ an architectural description language specified in xADL [2005] to describe OAs that can be designed and analyzed with a software architecture design environment [Medvidovic 1999], such as ArchStudio4 [2006]. We have taken this environment and extended it with a Software Architecture License Traceability Analysis module [cf. Asuncion 2008]. This allows for the specification of licenses as a list of attributes (license tuples) using a form-based user interface, similar to those already used and known for ArchStudio4 and xADL [ArchStudio 2006, Medvidovic 1999].

Figure 3 shows a screenshot of an ArchStudio4 session in which we have modeled the OA seen in Figure 1. OA software components, each of which has an associated license, are indicated by darker shaded boxes. Light shaded boxes indicate connectors. Architectural connectors may or may not have associated license information; those with licenses (such as archi-

tectural connectors that represent functional code) are treated as components during license traceability analysis. A directed line segment indicates a link. Links connect interfaces between the components and connectors. Furthermore, the Mozilla component as shown here contains a hypothetical subarchitecture for modeling the role of intra-application scripting, as might be useful in specifying license constraints for Rich Internet Applications. This subarchitecture is specified in the same manner as the overall system architecture, and is visible in Figure 5. The automated environment allows for tracing and analysis of license attributes and conflicts.

Figure 4 shows a view of the internal XML representation of a software license. Analysis and calculations of rights, obligations, and conflicts for the OA are done in this form. This schematic representation is similar in spirit to that used for specifying and analyzing privacy and security regulations associated with certain software systems [Breaux and Anton 2008].

With this basis to build on, it is now possible to analyze the alignment of rights and obligations for the overall system:

1. Propagation of reciprocal obligations

Reciprocal obligations are imposed by the license of a GPL’d component on any other component that is part of the same “work based on the Program” (i.e. on the first component), as defined in GPL. We follow the widely-accepted interpretation that build-time static linkage propagate the reciprocal obligations, but the “license firewalls” do not. Analysis begins, therefore, by propagating these obligations along all connectors that are not license firewalls.

2. Obligation conflicts

An obligation can conflict with another obligation contrary to it, or with the set of available rights, by requiring a copyright right that has not been granted. For instance, the Corel proprietary license for the WordPerfect component, CTL (Corel Transactional License), may be taken to entail that a licensee must not redistribute source code. However, an OSS license, GPL, may state that a licensee must redistribute source code. Thus, the conflict appears in the modality of the two otherwise identical obligations, “must not” in CTL and “must” in GPL. A conflict on the same point could occur also between GPL and a component whose license fails to grant the right to distribute its source code.

This phase of the analysis is affected by the overall set of rights that are required. If conflicts arise involving the union of all obligations in all components' licenses, it may be possible to eliminate some conflicts by selecting a smaller set of rights, in which case only the obligations for those rights need be considered.

Figure 5 shows a screenshot in which the License Traceability Analysis module has identified obligation conflicts between the licenses of two pairs of components ("WordPerfect" and "Linux OS", and "GUIDisplayManager" and "GUIScriptInterpreter").

3. Rights and obligations calculations

The rights available for the entire system (use, copy, modify, etc.) then are calculated as the intersection of the sets of rights available for each component of the system.

The obligations required for the whole system then are the union of the specific obligations for each component that are associated with those rights. Examples of specific obligations are "Licensee must retain copyright notices in the binary form of `module.c`" or "Licensee must publish the source code of `component.java` version 1.2.3."

Figure 6 shows a report of the calculations for the hypothetical subarchitecture of the Mozilla component in our archetypal architecture, exhibiting an obligation conflict and the single copyright right (to run the system) that the prototype tool shows would be available for the subarchitecture as a whole if the conflict is resolved; a production tool would also list the rights (none) currently available.

If a conflict is found involving the obligations and rights of linked components, it is possible for the system architect to consider an alternative linking scheme, employing one or more connectors along the paths between the components that act as a license firewall, thereby mitigating or neutralizing the component-component license conflict. This means that the architecture and the environment together can determine what OA design best meets the problem at hand with available software components. Components with conflicting licenses do not need to be arbitrarily excluded, but instead may expand the range of possible architectural alternatives if the architect seeks such flexibility and choice.

At build-time (and later at run-time), many of the obligations can be tested and verified, for example that the binaries contain the appropriate notices for their li-

censes, and that the source files are present in the correct version on the Web. These tests can be generated from the internal list of obligations and run automatically. If the system's interface were extended to add a control for it, the tests could be run by a deployed system.

The prototype License Traceability Analysis module provides a proof-of-concept for this approach. We encoded the core provisions of four licenses in XML for the tool—GPL, MPL, CTL, and AFL (Academic Free License)—to examine the effectiveness of the license tuple encoding and the calculations based upon it. While it is clear that we could use a more complex and expressive structure for encoding licenses, in encoding the license provisions to date we found that the tuple representation was more expressive than needed; for example, the actor was always "licensee" and seems likely to remain so, and we found use for only three operations or modalities. At this writing, the module shows proof of concept for calculating with reciprocal obligations by propagating them to adjacent statically-linked modules; the extension to all paths not blocked by license firewalls is straightforward and is independent of the scheme and calculations described here. Reciprocal obligations are identified in the tool by lookup in a table, and the meaning and scope of reciprocity is hard-coded; this is not ideal, but we considered it acceptable since the legal definition in terms of the reciprocal licenses will not change frequently. We also focused on the design-time analysis and calculation rather than build- or run-time as it involves the widest range of issues, including representations, rights and obligations calculations, and design guidance derived from them.

Based on our analysis approach, it appears that the questions of what license (if any) covers a specific configured system, and what rights are available for the overall system (and what obligations are needed for them) are difficult to answer without automated license-architecture analysis. This is especially true if the system or sub-system is already in operational run-time form [cf. Kazman and Carrière 1999]. It might make distribution of a composite OA system somewhat problematic if people cannot understand what rights or obligations are associated with it. We offer the following considerations to help make this clear. For example, a Mozilla/Firefox Web browser covered by the MPL (or GPL or LGPL, in accordance with the Mozilla Tri-License) may download and run intra-application script code that is covered by a different license. If this script code is only invoked via dynamic run-time linkage, or via a client-server transaction protocol, then there is no propagation of license rights or obligations. However,

if the script code is integrated into the source code of the Web browser as persistent part of an application (e.g., as a plug-in), then it could be viewed as a configured sub-system that may need to be accessed for license transfer or conflict implications. Another different kind of example can be anticipated with application programs (like Web browsers, email clients, and word processors) that employ Rich Internet Applications or mashups entailing the use of content (e.g., textual character fonts or geographic maps) that is subject to copyright protection, if the content is embedded in and bundled with the scripted application sub-system. In such a case, the licenses involved may not be limited to OSS or proprietary software licenses.

In the end, it becomes clear that it is possible to automatically determine what rights or obligations are associated with a given system architecture at design-time, and whether it contains any license conflicts that might prevent proper access or use at build-time or run-time, given an approach such as ours.

6. Discussion

Software system configurations in OAs are intended to be adapted to incorporate new innovative software technologies that are not yet available. These system configurations will evolve and be refactored over time at ever increasing rates [Scacchi 2007], components will be patched and upgraded (perhaps with new license constraints), and inter-component connections will be rewired or remediated with new connector types. As such, sustaining the openness of a configured software system will become part of ongoing system support, analysis, and validation. This in turn may require ADLs to include OSS licensing properties on components, connectors, and overall system configuration, as well as in appropriate analysis tools [cf. Bass, Clements, and Kazman 2003, Medvidovic 1999].

Constructing these descriptions is an incremental addition to the development of the architectural design, or alternative architectural designs. But it is still time-consuming, and may present a somewhat daunting challenge for large pre-existing systems that were not originally modeled in our environment.

Advances in the identification and extraction of configured software elements at build time, and their restructuring into architectural descriptions is becoming an ever more automatable endeavor [cf. Choi 1990, Kazman 1999, Jansen 2008]. Further advances in such efforts have the potential to automatically produce architectural descriptions that can either be manually or semi-automatically annotated with their license con-

straints, and thus enable automated construction and assessment of build-time software system architectures.

The list of recognized OSS licenses is long and ever-growing, and as existing licenses are tested in the courts we can expect their interpretations to be clarified and perhaps altered; the GPL definition of “work based on the Program”, for example, may eventually be clarified in this way, possibly refining the scope of reciprocal obligations. Our expressions of license rights and obligations are for the most part compared for identical actors, actions, and objects, then by looking for “must not” in one and either “must” or “may” in the other, so that new licenses may be added by keeping equivalent rights or obligations expressed equivalently. Reciprocal obligations, however, are handled specially by hard-coded algorithms to traverse the scope of that obligation, so that addition of obligations with different scope, or the revision of the understanding of the scope of an existing obligation, requires development work. Possibly these issues will be clarified as we add more licenses to the tool and experiment with their application in OA contexts.

Lastly, our scheme for specifying software licenses offers the potential for the creation of shared repositories where these licenses can be accessed, studied, compared, modified, and redistributed.

7. Conclusion

The relationship between open architecture, open source software, and multiple software licenses is poorly understood. OSS is often viewed as primarily a source for low-cost/free software systems or software components. Thus, given the goal of realizing an OA strategy together with the use of OSS components and open APIs, it has been unclear how to best align software architecture, OSS, and software license regimes to achieve this goal. Subsequently, the central problem we examined in this paper was to identify principles of software architecture and software copyright licenses that facilitate or inhibit how best to insure the success of an OA strategy when OSS and open APIs are required or otherwise employed. In turn, we presented an analysis scheme and operational environment that demonstrates that an automated solution to this problem exists.

We have developed and demonstrated an operational environment that can automatically determine the overall license rights, obligations, and constraints associated with a configured system architecture whose components may have different software licenses. Such an environment requires the annotation of the participating software elements with their corresponding li-

censes. These annotated software architectural descriptions can be prescriptively analyzed at design-time as we have shown, or descriptively analyzed at build-time or run-time. Such a solution offers the potential for practical support in design-, build-, and run-time license conformance checking and the ever-more complex problem of developing large software systems from configurations of software elements that can evolve over time.

8. Acknowledgments

The research described in this report has been supported by grants #0808783 from the U.S. National Science Foundation, the Acquisition Research Program at the Naval Postgraduate School, and the Daegu Global R&D Collaboration Center, Daegu, Korea. No endorsement implied.

References

- Alspaugh, T.A and Antón, A.I., (2007). Scenario Support for Effective Requirements, Information and Software Technology, 50(3), 198-220.
- ArchStudio (2006). ArchStudio 4 Software and Systems Architecture Development Environment. Institute for Software Research, University of California, Irvine.
<http://www.isr.uci.edu/projects/archstudio/>
- Asuncion, H. (2008). Towards Practical Software Traceability, in Companion of the 30th Intern. Conf. Software Engineering, 1023-1026, Leipzig, Germany.
- Bass, L., Clements, P., and Kazman, R., (2003). Software Architecture in Practice, 2nd Edition, Addison-Wesley Professional, New York..
- Breaux, T.D. and Anton, A.I. (2008). Analyzing Regulatory Rules for Privacy and Security Requirements, IEEE Trans. Software Engineering, 34(1), 5-20.
- Choi, S. and Scacchi, W. (1990). Extracting and Restructuring the Design of Large Systems, IEEE Software, 7(1), 66-71.
- Feldt, K., (2007). Programming Firefox: Building Rich Internet Applications with XUL, O'Reilly Press, Sebastopol, CA.
- Fontana, R., Kuhn, B.M., Molgen, E., et al. (2008). A Legal Issues Primer for Open Source and Free Software Projects, Software Freedom Law Center, Version 1.5.1,
<http://www.softwarefreedom.org/resources/2008/foss-primer.pdf>
- Fielding, R. and Taylor, R.N., (2002). Principled Design of the Modern Web Architecture, ACM Transactions Internet Technology, 2(2), 115-150.
- Hohfeld, W.N. (1913). Some Fundamental Legal Conceptions as Applied in Judicial Reasoning. Yale Law Journal, 23(1), 16-59.
- Jansen, A., Bosch, J., and Avgeriou, P. (2008). Documenting After the Fact: Recovering Architectural Design Decisions, J. Systems and Software, 81(4), 536-557.
- Kazman, R. and Carrière, J. (1999). Playing Detective: Reconstructing Software Architecture from Available Evidence. J. Automated Software Engineering, 6(2), 107-138.
- Kuhl, F., Weatherly, R., and Dahmann, J., (2000). Creating Computer Simulation Systems: An Introduction to the High Level Architecture, Prentice-Hall PTR, Upper Saddle River, New Jersey.
- Medvidovic, N., Rosenblum, D.S., and Taylor, R.N. (1999). A Language and Environment for Architecture-Based Software Development and Evolution. In Proc. 21st Intern. Conf. Software Engineering (ICSE '99). 44-53, IEEE Computer Society. Los Angeles, CA.
- Meyers, B.C. and Obendorf, P., (2001). Managing Software Acquisition: Open Systems and COTS Products, Addison-Wesley, New York.
- Nelson L. and Churchill, E.F., (2006). Repurposing: Techniques for Reuse and Integration of Interactive Services, Proc. 2006 IEEE Intern. Conf. Information Reuse and Integration, September.
- OSI (2008). The Open Source Initiative,
<http://www.opensource.org/>
- Rosen, L. (2005). Open Source Licensing: Software Freedom and Intellectual Property Law, Prentice-Hall PTR, Upper Saddle River, New Jersey.
<http://www.rosenlaw.com/oslbook.htm>
- Scacchi, W., (2002). Understanding the Requirements for Developing Open Source Software Systems, IEE Proceedings--Software, 149(1), 24-39, February.
- Scacchi, W. (2007). Free/Open Source Software Development: Recent Research Results and Emerging Opportunities, Proc. European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, Dubrovnik, Croatia, 459-468.
- Scacchi, W. and Alspaugh, T.A. (2008). Emerging Issues in the Acquisition of Open Source Software within the U.S. Department of Defense, Proc. 5th Annual Acquisition Research Symposium, Vol. 1, 230-244, NPS-AM-08-036, Naval Postgraduate School, Monterey, CA
- St. Laurent, A.M., (2004). Understanding Open Source and Free Software Licensing, O'Reilly Press, Sebastopol, CA.

Ven, K. and Mannaert, H., (2008). Challenges and Strategies in the Use of Open Source Software by Independent Software Vendors, Information and Software Technology, 50, 991-1002.

Wheeler, D.A., (2007). Open Source Software (OSS) in U.S. Government Acquisitions, The DoD Software Tech News, 10(2), 7-13, June.

xADL (2005). xADL 2.0: Highly-extensible architecture description language for software and systems. Institute for Software Research, University of California, Irvine.
<http://www.isr.uci.edu/projects/xarchuci/>

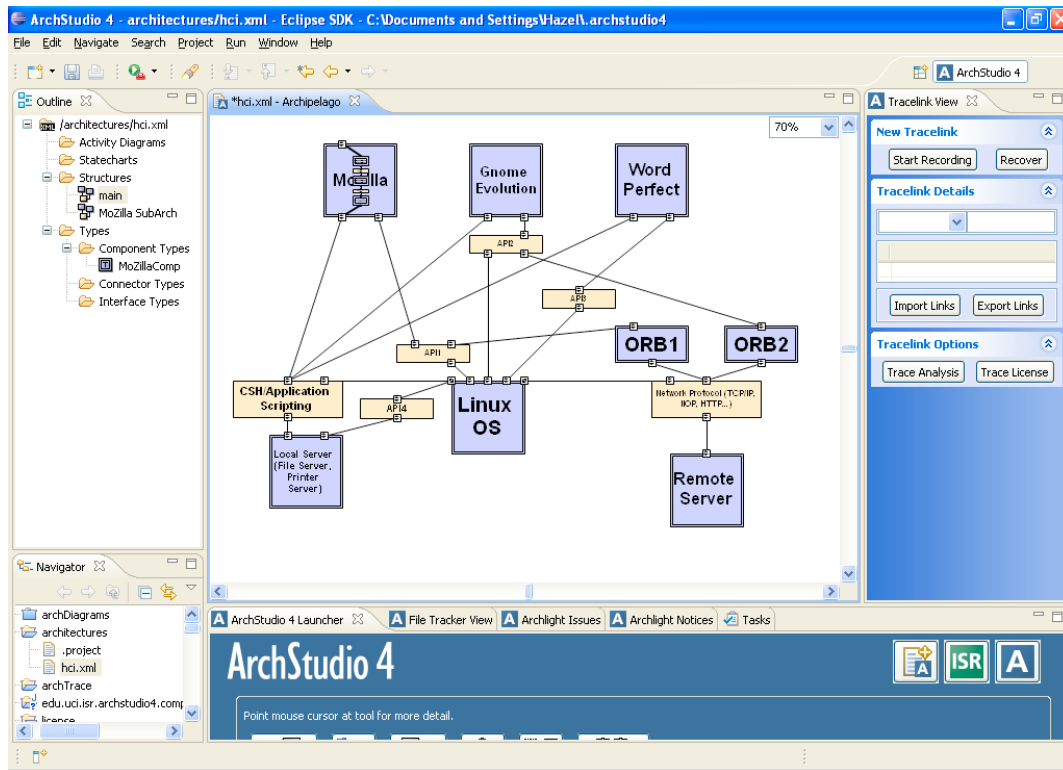


Figure 3. An ArchStudio 4 model of the open software architecture of Figure 1

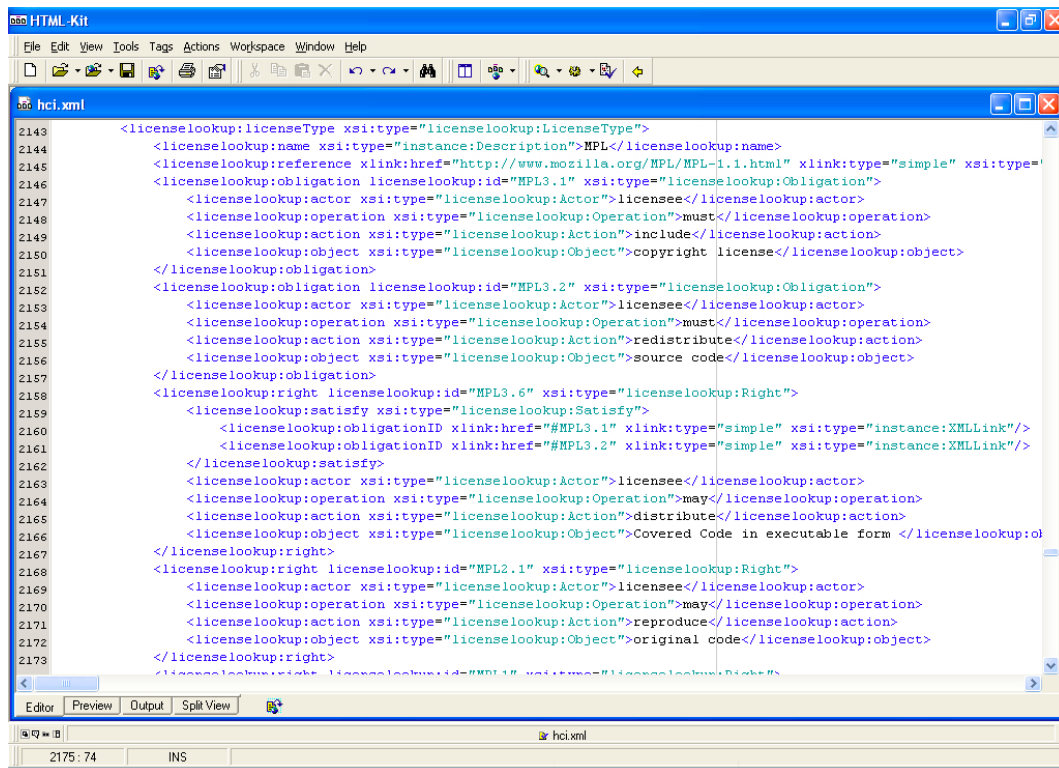


Figure 4. A view of the internal schematic representation of the Mozilla Public License

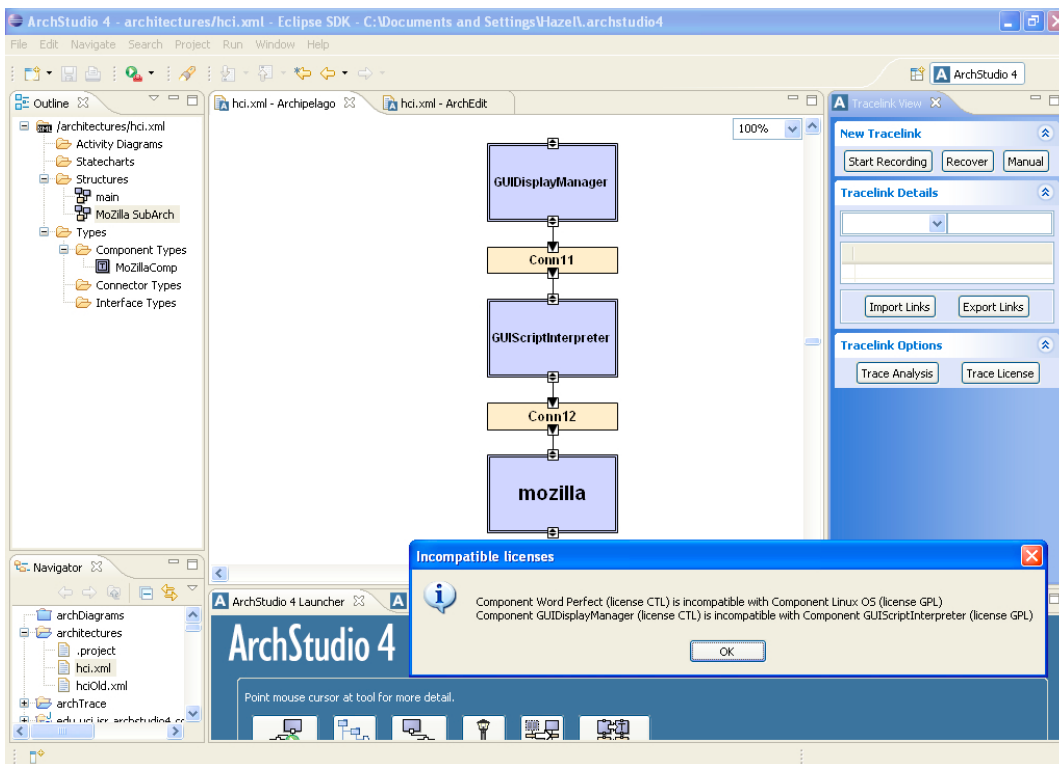
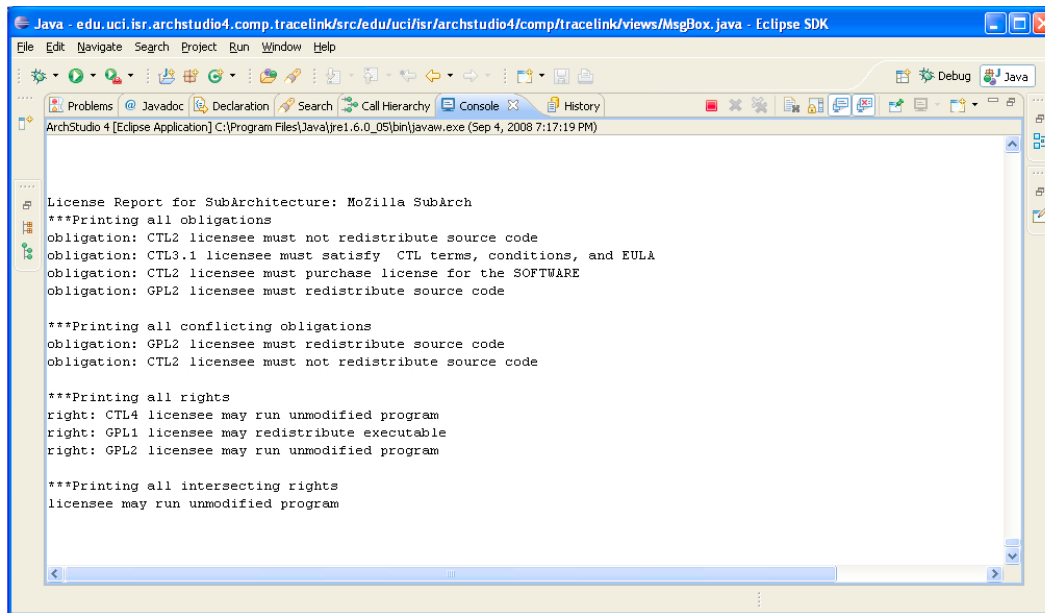


Figure 5. License conflicts have been identified between two pairs of components



The screenshot shows the Eclipse IDE's console window. The title bar reads "Java - edu.uci.isr.archstudio4.comp.tracelink/src/edu/uci/isr/archstudio4/comp/tracelink/views/MsgBox.java - Eclipse SDK". The console output is as follows:

```
License Report for SubArchitecture: Mozilla SubArch
***Printing all obligations
obligation: CTL2 licensee must not redistribute source code
obligation: CTL3.1 licensee must satisfy CTL terms, conditions, and EULA
obligation: CTL2 licensee must purchase license for the SOFTWARE
obligation: GPL2 licensee must redistribute source code

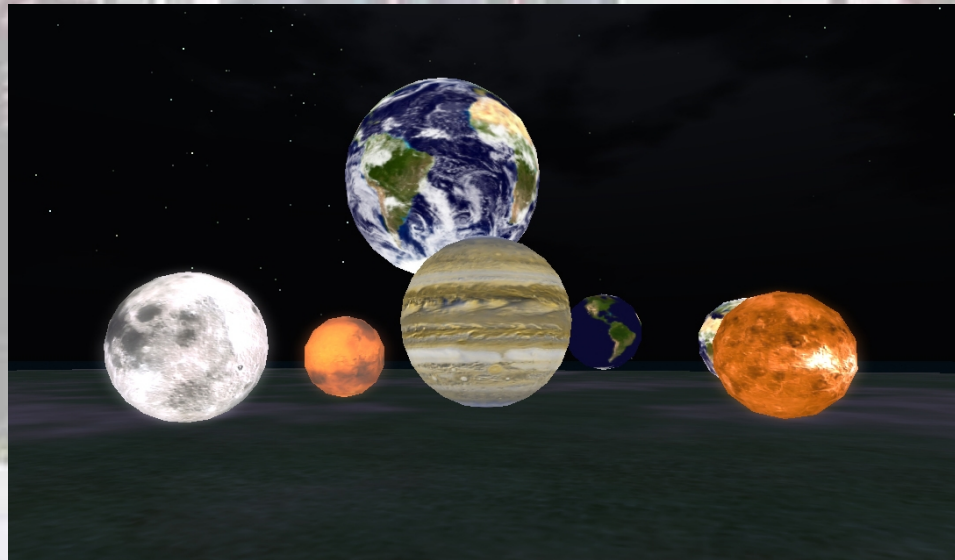
***Printing all conflicting obligations
obligation: GPL2 licensee must redistribute source code
obligation: CTL2 licensee must not redistribute source code

***Printing all rights
right: CTL4 licensee may run unmodified program
right: GPL1 licensee may redistribute executable
right: GPL2 licensee may run unmodified program

***Printing all intersecting rights
licensee may run unmodified program
```

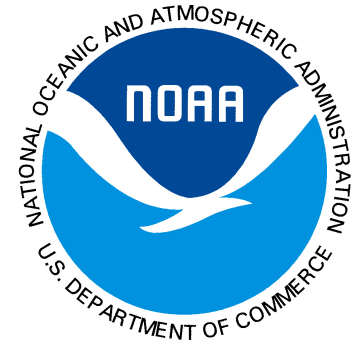
Figure 6. A report identifying the obligations, conflicts, and rights for the architectural model

Spherical Visualizations in Virtual Worlds



Spherical Displays in Virtual Worlds

We are experimenting with datasets from the
National Oceanic and Atmospheric Administration
(NOAA)'s Science on a Sphere (SOS) program.



Science On a Sphere Datasets



Science on a Sphere (SOS)



Science on a Sphere is a room sized, global display system that uses computers and video projectors to display planetary data onto a six foot diameter sphere, analogous to a giant animated globe.

Researchers at NOAA developed SOS as an educational tool to help illustrate Earth System science in a way that is both intuitive and captivating and to increase public understanding of the environment.

It is used as an instrument to enhance informal educational programs in science centers, universities, and museums across the world. It is currently installed at 47 sites worldwide.

<http://sos.noaa.gov/>

Discovery Science Center, Santa Ana, California



We are partnering with the Discovery Science Center, a local science museum in Santa Ana, which recently completed its own SOS installation.

Our goal is to bring the NOAA datasets to a broader audience using virtual worlds. We are looking to create a self-contained, open source solution that can easily be brought into the classroom.



SOS SphereCasting

An additional goal is to be able to receive SphereCasts from within a virtual world.



Viewing the first international SphereCast, live from COP15, the United Nations ClimateChange Conference in Copenhagen, Denmark from the Discovery Science Center in Santa Ana, CA.

What is SphereCasting?

A SphereCast is an SOS presentation done simultaneously at multiple sites by a single presenter, via the Internet. Many sites can receive the SphereCast, but only one site is the host. There are two components to a SphereCast: remote control of a presentation on an SOS system, and a live video (or audio) lecture that accompanies the SOS presentation.

The SphereCast is setup with a client-server architecture where the server is a specialized site (usually from NOAA headquarters) but any SOS installation can register to be a client/receiver of the SphereCast. Any commands the presenter issues to the SOS system at the host site are immediately replicated on all the SOS client systems. When the presenter loads a new dataset, that data is loaded on all the watching systems. When the presenter uses the remote control to start or stop animation, or orient the sphere, all the remote spheres behave identically.

All datasets must be pre-loaded in each client site. The streaming video component uses Apple Quicktime.

SOS Datasets/Playlists

- SOS Datasets include:

 - Images

 - Video

 - Animations

 - PIP (picture in picture overlays)

 - Audio

 - Slide presentations

 - Displayed on wall mounted displays, not on the SOS sphere

- SOS Playlists

 - Playlists organize and group together content for a presentation.

 - They can define the tilt and/or rotation of the sphere

 - They are simple text files that are read and interpreted by the main SOS application interface.

 - Playlists directly access the files in the datasets

OpenSim Platform

- We are experimenting with visualizing the SOS datasets in OpenSim.
- **OpenSim** is an open source server platform for hosting virtual worlds. It is compatible with the **Second Life** client.
- For the client, we are using the open source **Hippo** OpenSim viewer. It is compatible with the Second Life grid while taking advantage of OpenSim features and extensions.
- An OpenSim platform allows us to create a self-contained open source solution that can easily run on a portable computer.

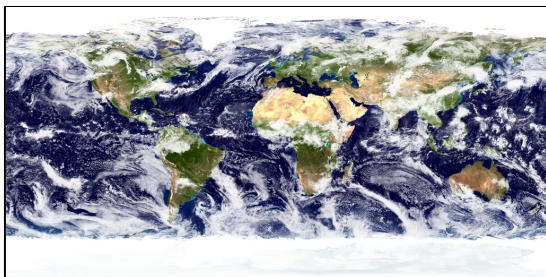
Goals

1. Display the SOS datasets in a virtual world using OpenSim.
2. Allow avatars to control which datasets are displayed.
3. Control the datasets from an external application.
4. Interpret SOS playlists and use them to control the datasets from an external app.

Datasets in OpenSim

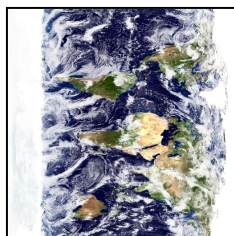
- Images

SOS images and video must have a 2:1 aspect ratio.

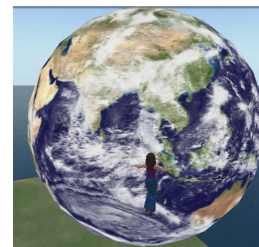


This “Blue Marble” image is an example image From the SOS dataset.

Images must be imported into OpenSim as textures that can be applied to the spherical prim object that implements the SOS sphere.



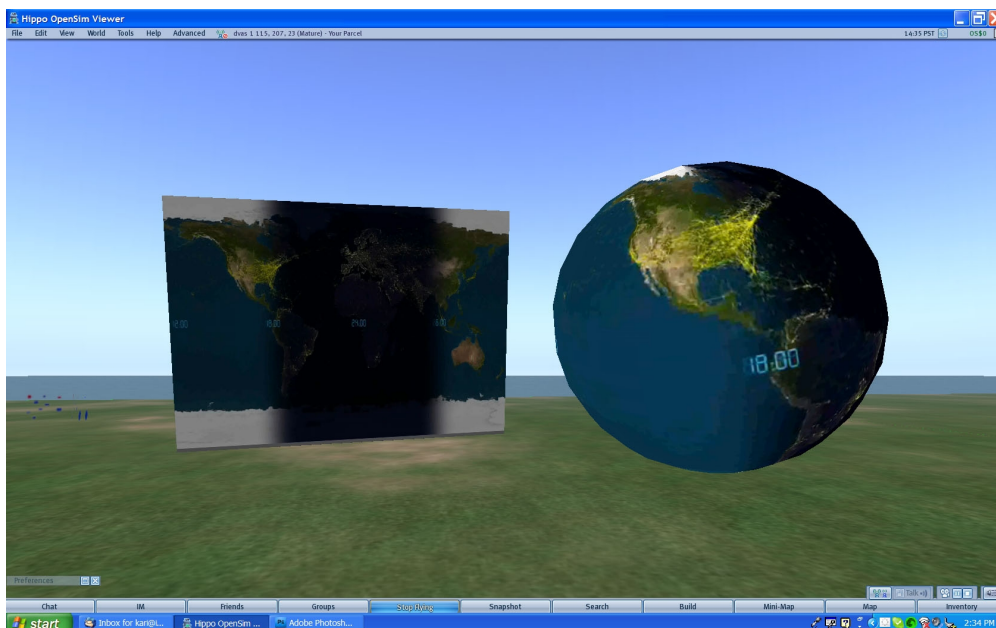
SOS images must be converted to 512X512 pixels and rotated before then can be imported as textures for the sphere (left). Note that the distortion created by scaling the image to a square format is counteracted when it is “stretched” around a spherical prim (right).



Datasets in OpenSim

- Video

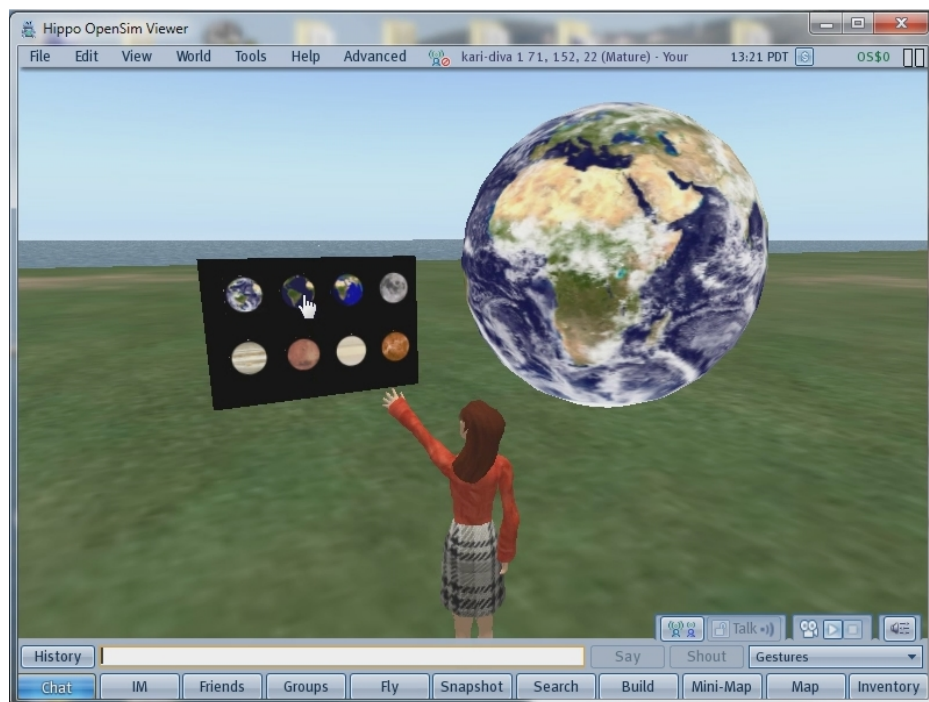
In OpenSim, mp4 video can be streamed from an external URL onto any object covered with a predefined “media texture”.



Here a video showing international aviation flight paths relative to time is being streamed simultaneously to both a rectangular and a spherical prim. The video is adjusted to wrap seamlessly around the sphere.

Controlling Datasets

- Using an avatar



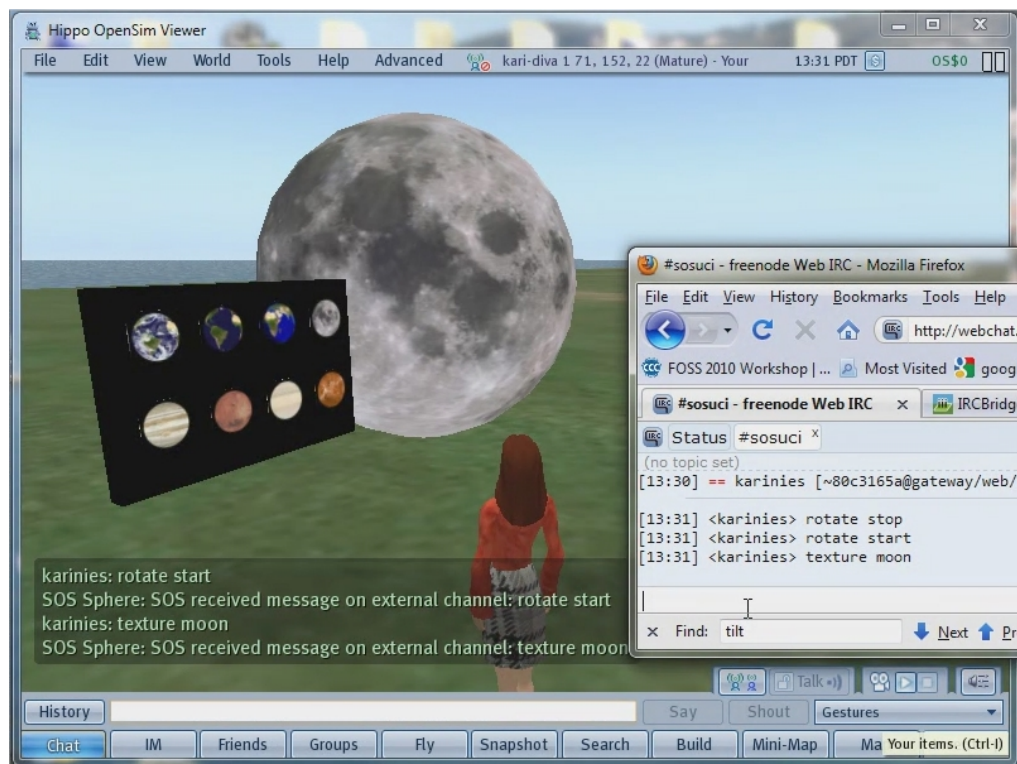
Here an avatar controls which dataset is displayed by “touching” a kiosk with a variety of planet display options.

The avatar can also “touch” the sphere to start and stop its rotation.

The touch events are detected and handled via scripts (written in LSL-Linden Scripting Language). Communication between objects is via internal chat channels.

Controlling Datasets

- From an external chat channel

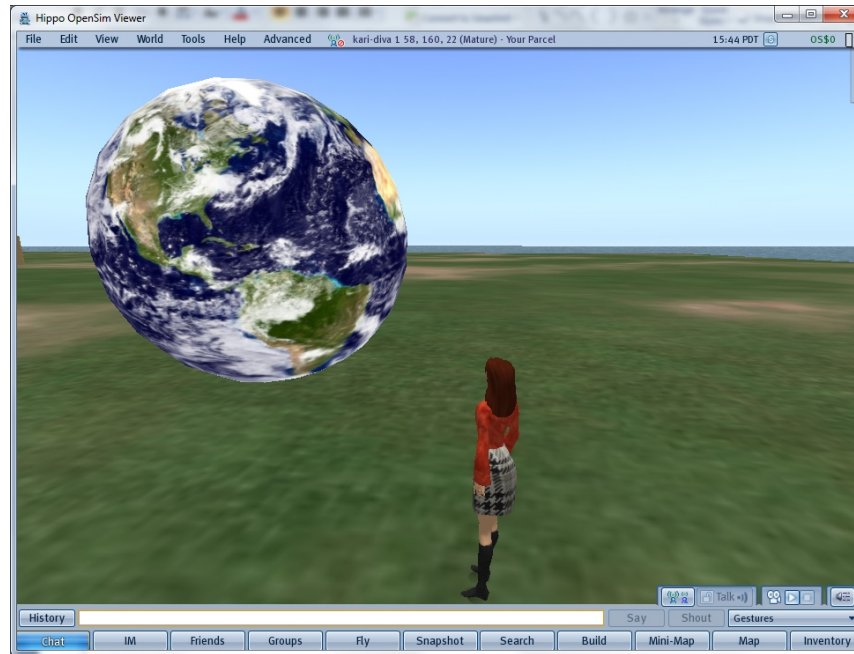


Here the client has been configured to listen to an external chat channel on freenode.net.

The user can send commands from the external channel. Here a user is controlling rotation and changing the sphere texture remotely.

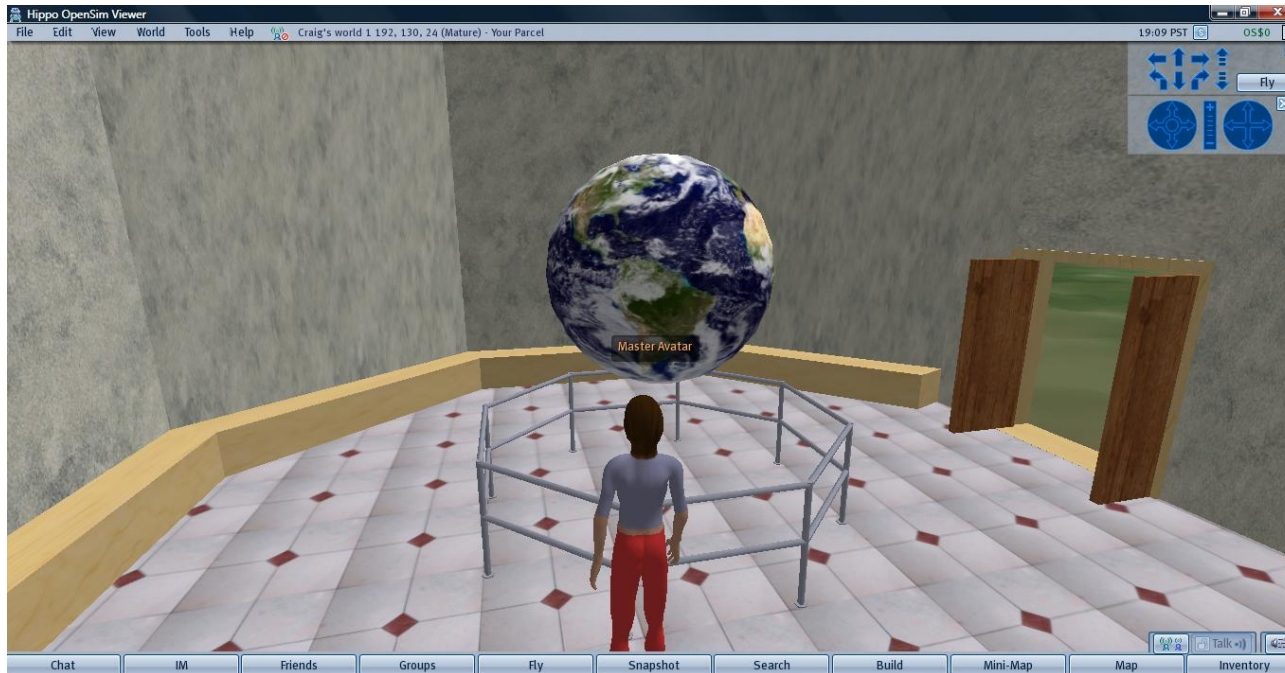
Tilt

- Scripts now support displaying the sphere at a specified x-axis tilt.



The SOS datasets display earth images at a 23.5 deg. tilt on the x-axis.

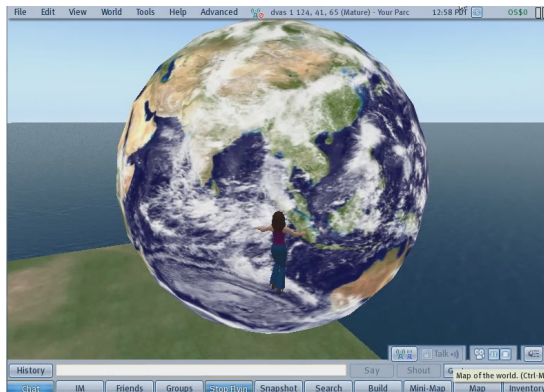
Modeling an SOS Installation



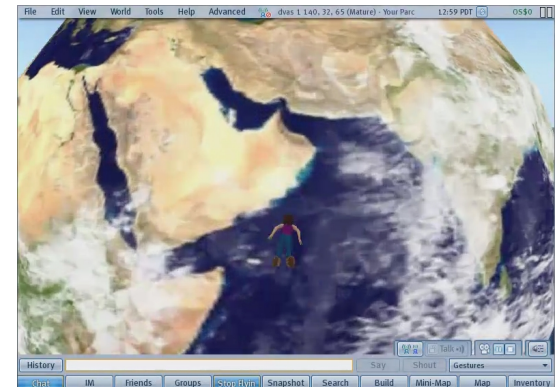
We are in the early stages of modeling the Discovery Science Center's SOS Installation in OpenSim.

More Spherical Visualizations

- Hollowed Sphere



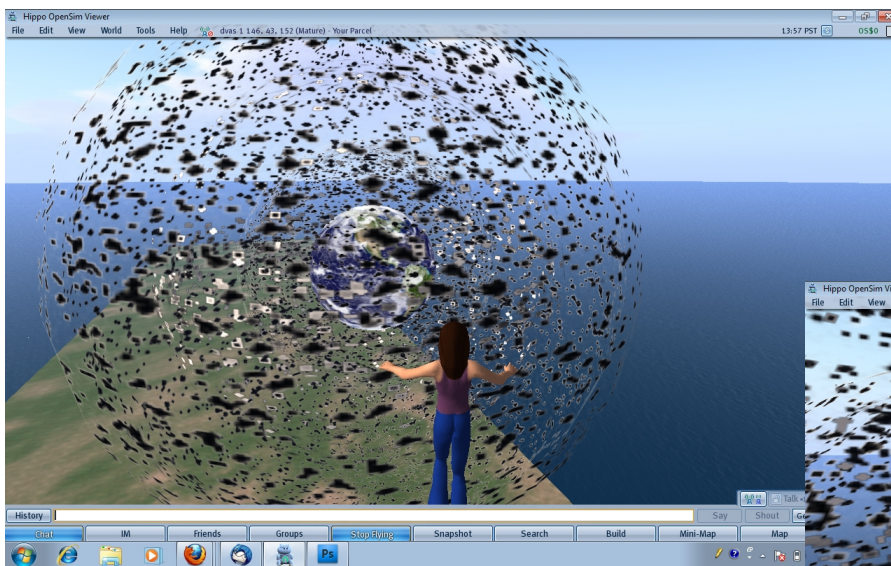
Here an avatar is shown entering a hollow sphere that contains a smaller sphere in it's core. Both are textured with SOS data images.



In this view the Avatar has entered the interior of the larger sphere and is looking outward. Note that the texture image is visible for the spheres interior and also that the image is a reversed from the inside looking out.

More Spherical Visualizations

- Space debris visualization using hollowed spheres



We've been exploring visualizations of space debris in virtual worlds.

Here we've simulated a visualization of debris in the earth's atmosphere. Surrounding the earth textured sphere in the center are three concentric hollowed spheres with transparent textures.



OutRun: Pervasive Games and Designing the De-Simulation of Eight-Bit Driving

Garnet Hertz

Center for Computer Games and Virtual Worlds
Institute for Software Research
University of California, Irvine
209 Information and Computer Science 2
Irvine, CA 92697, USA

ghertz@uci.edu (<http://conceptlab.com/>)

ABSTRACT

This paper outlines the development process of a mixed reality video game prototype that combines a classic arcade driving game with a real world vehicle. In this project the user, or player, maneuvers the car-shaped arcade cabinet through actual physical space using a screen as a navigational guide which renders the real world in the style of an 8-bit video game. This case study is presented as a “pervasive game”: an attempt to disrupt the everyday by highlighting and inverting conventional behavior through humor and paradox.

Categories and Subject Descriptors

H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems – *artificial, augmented and virtual realities*; H.5.m [Simulation and Modeling]: Types of Simulation – *gaming*; J.5 [Arts and Humanities]: Fine Arts, Performing Arts; H.5.m [Information Interfaces and Presentation]: Miscellaneous.

General Terms

Design, Experimentation, Human Factors, Theory.

Keywords

Game design, game innovation, mixed reality, augmented reality, alternate reality, pervasive gaming, live action role playing, electronic art, experimental interfaces, prototyping.

1. INTRODUCTION

The OutRun project is a game and media art project that explores the overlap between the physical world and game environments. OutRun explores the act of driving a vehicle and the interstitial space between everyday life (driving an automobile) and a simulation of it (playing a driving video game) by combining the real world and OutRun, an eight-bit arcade driving game released by Sega in 1986. This project features two main components:

1. **Cabinet-Car:** A car-shaped sit-down arcade cabinet from Sega's OutRun is converted into a small car that can actually drive. This is done by modifying an existing fiberglass and wood cabinet with motors, wheels and drivetrain components from an electric golf cart. The original arcade cabinet is modeled after a 1984 Ferrari Testarossa. This customized cabinet-car will use the existing videogame controls (steering wheel, acceleration pedal,

brake) to control the vehicle. It is expected that the maximum speed of the car will be no more than 20 miles (32 kilometers) per hour. See Figure 1 for a mockup diagram of the completed cabinet-car.

2. **Custom Augmented Reality Software:** The screen, which is positioned in front of the driver, renders the real world in the style of the 1986 video game OutRun. This is done through custom-built computer vision software and GPS sensors that calculate the location of the experimental vehicle and display a street-level view rendered in the style of the vintage video game. In other words, the driver only sees an eight-bit-style game rendered in their “windshield,” which appears as if they are playing the 1986 videogame. Accelerating or turning the car-cabinet in the real world will proportionally change the display. Although the screen will mimic the real world around it, it is expected that the augmented display and the real world will not match perfectly.



Figure 1. The proposed OutRun cabinet-car.

This project is motivated by the following concepts:

1. **The De-Simulation of Driving** - This project de-simulates the driving component of a videogame. Driving game simulations strive to be increasingly realistic, but this realism is usually focused on graphical representations. Instead, this system pursues “real” driving through a videogame as its primary goal.

2. **GPS Navigation & Mixed Reality Parallax** - Driving in a real automobile with a GPS navigation system can be game-like. This project explores the consequences of only using only a computer

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DRAFT: NOT FOR CITATION OR DISTRIBUTION

model of the world as a navigation tool for driving. The windshield of this project's vehicle only shows a re-rendered simulation of the immediate environment, and as a result, driving it in the real world is often difficult or dangerous. As a result, this project explores and investigates how augmented reality and GPS data differs from the physical world, and what happens when an augmentation of reality envelops and obfuscates reality.

2. CONTEXTS IN GAME CULTURE

The OutRun project shares similarities to several existing genres of game design, namely pervasive games (including live-action role-playing games, alternate reality games, and big games) and driving simulators.

Pervasive games “move beyond the traditional computer interfaces and into the physical world to occupy time and place on a human scale.”[Falk, Jennica; Davenport, Glorianna (2004). "Live Role-Playing Games: Implications for Pervasive Gaming" (PDF). Entertainment Computing – ICEC 2004. Lecture Notes in Computer Science. 3166. Springer Berlin / Heidelberg. page 127.] As Falk and Davenport describe, pervasive games blur lines of demarcation between games and life: players are mixed with game characters, the real world is overlapped with game worlds, and game artifacts are combined with real world objects. Contemporary pervasive games consist of at least three subgenres: live action role-playing games, alternate reality games and big games.

Live action role-playing (LARP) games are pervasive role-playing games where individuals physically act out their characters' actions, typically in a fictional setting that is represented by the real world. Players embody their character roles through costumes and physical actions, maintaining a clear sense of being in-character or out-of-character [Tychsen et al., Live Action Role Playing Games, page 255]. Alternate Reality games, on the other hand, are pervasive games that attempt to delete the line between being in-character and out-of-character: “they do everything in their power to erase game boundaries – physical, temporal and social – and to obscure the metacommunications that might otherwise announce, 'This is play.'”[Jane McGonigal, A Real Little Game: The Performance of Belief in Pervasive Play]. Instead of fictional character play, alternate reality games have an aesthetic of trying to not look like either a game or a fantasy. Inspiration for alternate reality games is primarily driven by an extension of gaming environments through mobile network technologies and ubiquitous computing. [Jane McGonigal, A Real Little Game: The Performance of Belief in Pervasive Play].

Within the category of pervasive games, the OutRun project perhaps has the most similarity with some styles of “Big Games,” especially the Pac-Manhattan project led by Frank Lantz at NYU's Interactive Telecommunications graduate program in 2004 (Figure 2). Pac-Manhattan is a large-scale urban game that utilized the New York City grid to recreate the 1980's video game Pac-Man. A player dressed as Pac-Man runs around an area of Manhattan while attempting to collect all of the virtual “dots” that ran the length of the streets; four players dressed as the ghosts Inky, Blinky, Pinky and Clyde attempted to catch Pac-Man before all of the dots were collected [Pac-Manhattan, <http://pacmanhattan.com/about.php>, 2004].



Figure 2. Pac-Manhattan, a “big” game developed in 2004 where individuals play a Pac-Man inspired physical game through the streets of Manhattan.

Like the OutRun project, Pac-Manhattan takes an existing eight-bit game and explores what happens when it is mapped into the larger “real world” of buildings, streets, and cities. In comparison to LARP or Alternate Reality games, Big Games like Pac-Manhattan have a simple premise: take an existing video game and explore what happens when it is scaled up into the real world. OutRun does this, but also does the inverse: it attempts to take the real world and translate it into a “small game” on a computer screen.



Figure 3. The F1Showcar, a full size motion simulator computer game car system.

The OutRun project also borrows from non-pervasive game systems in its concept. The contemporary racing simulator video game community, for example, has developed a substantial selection of customized computer furniture and hardware peripherals to augment its genre of game play. Basic systems include foot pedal and steering wheel peripherals, while more involved configurations include full scale car cockpits and hydraulic motion tables that simulate the physics of real-world driving (Figure 3) [F1 ShowCar, <http://f1showcar.com/>].

The OutRun project extends this pursuit of immersive driving simulation by reconnecting the reproduction with its original source: actually driving. In the process, OutRun borrows from pervasive gaming by using the real world as its game space.

The end result explores the relatively new phenomenon of driving an automobile and simultaneously looking at a computer screen. Contemporary driving increasingly involves computing: using a GPS navigation system, in-dash entertainment, or text messaging while driving are progressively part of real-world driving culture. Although automobile GPS navigation is not considered a game, malfunctioning wayfinding algorithms and inaccuracies in street-level data can produce a humorous, frustrating or dangerous “pervasive game.” These overlaps have serious consequences and deserve to be explored: in 2008, the British newspaper *The Mirror* estimated that automobile GPS navigation systems have caused 300,000 accidents in the UK [The Mirror, SatNav danger revealed: Navigation device blamed for causing 300,000 crashes, Tanith Carey, 21/07/2008]. OutRun is not proposing a solution to precarious uses of computers in vehicles, but intentionally magnifying and embracing the problem. In the process, the system strives to function in the historical role of a trickster: to disrupt the everyday by highlighting and inverting conventional behavior through humor and paradox. [Hertz, Un-Simulations, Tricksters and Radical Thought, Blackflash Magazine, 2009] OutRun extends simulation and perverts it at the same time: it is like a fantasy taken to its extreme and gone humorously wrong. Being a clown doesn't accomplish much, but trying to intelligently invert social conventions and assumptions can constructively and artistically rewire our understandings of who we are.

3. PROJECT DESIGN AND DEVELOPMENT PROCESS

The OutRun system began as a project proposal in January 2009. Active development started in June 2009 after five undergraduate students at UC Irvine were recruited as interns from the course “Computer Games as Art, Culture and Technology” [Losh, DAC09, 2009]. I oversaw development production, and the team was split up into the following general areas – Hardware: Matt Wong and Erik Olson; Game and Visualization Software: Chris Guevara; Media Assets: David Dinh; and Documentation: Richard Vu. The Laboratory for Ubiquitous Computing and Interaction at UC Irvine provided intern office space and software licenses, while the Arts Computation Engineering program at UCI provided large format studio space for the physical construction of the cabinet-car.

3.1 Software Design and Development Process



Figure 4. A screenshot from the first level of the original 1986 video game OutRun.

The original OutRun game featured a pixelated 1984 Ferrari Testarossa that the user controlled down winding roads through a number of different landscapes. For the purposes of this project, only the first level will be used as a source of inspiration: it is an idealized version of a California-esque beach community with palm trees, windsurfers and surf shops – plus a six lane freeway barreling through it all (Figure 4).

Software development of the project to date has focused on building a software system that is able to render the real world as it would be drawn in the original OutRun game. In other words, the software's key task was to draw the world like an eight-bit driving videogame: if a road actually appears in front of you that curves to the left, the system would draw an eight-bit style road on the screen that curved to the left.

Development of this software has taken two different paths: 1. Using computer vision to detect objects in the real world, and 2. Using location-aware technologies like global positioning systems (GPS) to position the user in a virtual world.

3.1.1 Computer Vision Development Path

Development of the OutRun software began with the development of a realtime computer vision system using Max/MSP/Jitter 5, an interactive graphical programming environment for music, audio, and media [Cycling '74, Max/MSP/Jitter, <http://cycling74.com/>]. Jitter – an architecture optimized for use with video – was used to take video data and store it in a two dimensional array. Cvjit, a collection of of Max/MSP/Jitter computer vision tools by Jean-Marc Pelletier, was used for its ability to detect line edges in video data. Along with pre-defined Max/MSP/Jitter objects and a line detection tool in cv.jit, we created our own computer vision objects in Java to tackle what we saw as our main problem: to detect the shape of roads.



Figure 5. Driving video processed using Jean-Marc Pelletier's cv.jit.lines, a Max/MSP/Jitter tool for detecting line edges in video data.

Cvjit detected numerous lines in a scene (Figure 5) and to focus on roads, we developed a number of custom filtering algorithms to remove lines that we thought weren't road-like. Then, by calculating the convergence point of the road-like lines, we tried to determine the general direction of the road (Figure 6).



Figure 6. Driving video processed with cv.jit.lines and custom filtering modules. The dot in the center of the image shows the calculated direction of the road, which is the average convergence point of filtered lines.

This approach has been useful in programmatically determining the direction of real-world roads, but is far from perfect. The current system is only set to detect a single road endpoint, does not calculate road width, and gets confused by shadows, crosswalks and other variables.

Despite these problems, the calculated endpoint produces interesting results when rendered in an eight-bit driving video game style. To do this, we constructed a simple Adobe Flash-based driving game that was controllable by our Max/MSP/Jitter vision software. We used Antony Dzeryn's Retro Racer as a starting point for this development, a Flash-based open source driving game that was similar in style to the original OutRun game. By modifying the ActionScript and replacing media assets, we modified Retro Racer to look more like OutRun and enabled communication between Max/MSP/Jitter and Flash (Figure 7).



Figure 7. Custom Flash-based game developed in the style of OutRun, with road endpoint controllable by computer vision data in Max/MSP/Jitter.

3.1.2 Locative / GPS Development Path

The second development path that is being pursued for this project is through location-aware technologies like global positioning systems (GPS) to position the user in an eight-bit style virtual world. This development path has been pursued with undergraduate students Matt Shigekawa, Jesse Joseph and Mike Tang starting in September 2009.

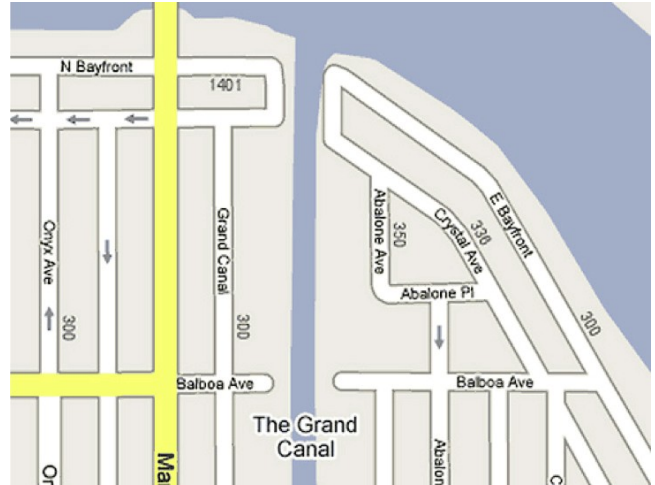


Figure 8. A standard Google Map view of Balboa Island, California that shows street data and land outlines.



Figure 9. Street data and land outlines of Balboa Island drawn in an eight-bit videogame style.

This approach takes data from the physical world – like street data, land shapes and building placement – and builds a pre-rendered three dimensional virtual world that can be navigated by using a real time GPS sensor that controls a handheld display.

The three dimensional virtual world was built to be displayed in Google Earth, and was constructed in Keyhole Markup Language (KML), an XML-based language schema for expressing geographic annotation and visualization on two-dimensional maps and three-dimensional Earth browsers [Google, KML – Google Code, <http://code.google.com/apis/kml/>]. To simplify development and testing of this approach, we focused on small geographic areas: Balboa Island and the UC Irvine Campus in

Orange County, California. Ground maps for this eight-bit-styled environment were built non-programmatically from street and earth data (Figures 8 and 9), and objects – like trees, buildings, automobiles and people – were taken from the original OutRun game. The world was assembled using Google SketchUp, a 3D modeling program that facilitates the placement of models in Google Earth (Figures 10 and 11) [Google SketchUp, <http://www.sketchup.com>].

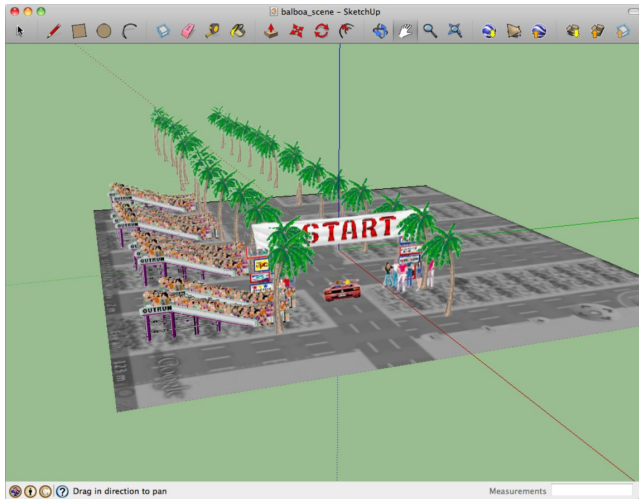


Figure 10. Two dimensional media assets being assembled using Google SketchUp, recreating the start sequence of the original OutRun game on Park Avenue, Balboa Island.

After assembling scenes on Balboa Island and the UC Irvine Campus the locations were physically navigated with a Q1UP-XP Samsung UMPC Tablet and a Nokia LD-1W Wireless GPS Module, with position data fed from the GPS module to Google Earth by GooPs Pro, a Google Earth GPS Tracking and Navigation tool.



Figure 11. View down Park Avenue of Balboa Island in Google Earth with placed OutRun-style media assets.

Real-world testing of this locative/GPS system showed that the Nokia LD-1W Wireless GPS module had a resolution of approximately ten to fifteen meters and a location refresh rate of

about three seconds. These factors made real-time interaction with the virtual world not as immersive as we would have liked: in Balboa Island tests, the lack of GPS resolution positioned us as driving through a row of trees and the position refresh rate produced a lurching/hopping sensation as you moved down the street, at least at the perspective of the original video game. A locative development path will likely require a high resolution GPS receiver (like a Navcom SF-2050G with an accuracy of 10cm) and tweening between location waypoints.

The sensation of speed was also considerably different from the original game: driving a real-world car at 40 kilometers per hour through the eight-bit virtual world, for example, seemed painfully slow in comparison to the 200+ km/h average speed of the original game car. In our mixed reality GPS prototype it felt as if you were crawling through the game world, highlighting that in everyday driving one normally doesn't race a car down residential streets. This gap between the expectations of the game world and action in the real world could have some interesting implications for gameplay: it could either encourage players to drive recklessly in the real world to recreate the original eight-bit game experience, or highlight how slow and mundane everyday real-world driving is in comparison to a driving game.

3.2 Cabinet-Car Design and Development Process

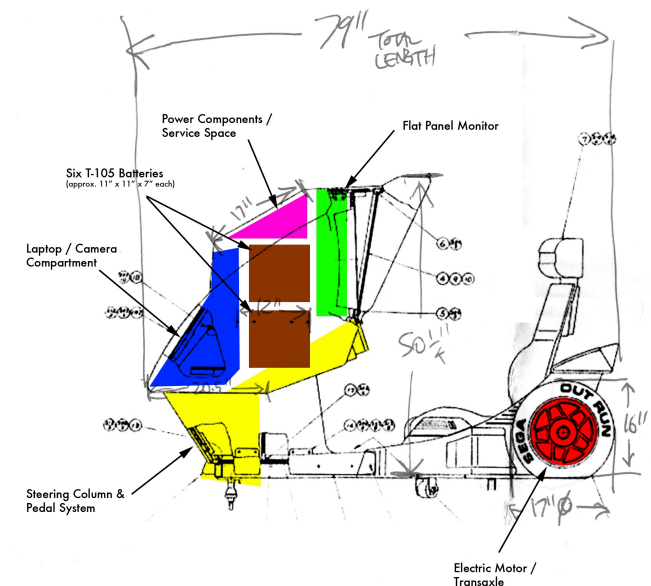


Figure 12. Proposed Cabinet-Car Component Layout, using side profile of video game cabinet as a reference.

The cabinet-car is the conglomeration of two components: a sit-down video game arcade cabinet and an electric golf cart (Figure 12). The arcade cabinet chosen was the “deluxe” OutRun configuration offered by Sega, weighing with 350 kg (770 lbs) with hydraulic side-to-side motion, a large fiberglass driving cockpit and a 25 inch CRT monitor. Our cabinet was purchased from a private party in Maine (USA) in July 2009 (Figure 13).



Figure 13. Original OutRun "Deluxe" arcade game cabinet (background) and 1959 Turfrider Mark IV golf cart (foreground).

In August 2009 a three-wheeled golf cart was purchased as a drivetrain for the system: a 1959 Turfrider Mark IV (Figure 13). The Mark IV golf cart was selected due to its three wheel configuration, unique "googie" space-age styling and similar form factor to the OutRun cabinet. However, after testing the system it lacked efficiency, power, stability and the tricycle steering mechanism would be difficult to interface with the arcade steering wheel. As a result, the classic golf cart was abandoned in favor of a four-wheeled 2007 Ez-Go RXV, which grants at the Institute for Software Research supported since Fall 2009. The Ez-Go RXV electric golf cart provides a more functional drivetrain, with greater reliability, power, battery duration and serviceability to make the make the system more viable as a robust exhibit/vehicle device.

4. CONCLUSION: PERVERSIVE GAMING

In 2003 Jane McGonigal humorously raised the idea that the combination of pervasive and immersive play results in "pervasive" gaming: where "players are prone to falling for the games' dissimulative rhetoric [...] They wind up believing in their play too much for their own good." [Jane McGonigal, "A Real Little Game: The Performance of Belief in Pervasive Play." Digital Games Research Association (DiGRA) "Level Up" Conference Proceedings. November 2003.] Through the OutRun project, I'd like to return to McGonigal's concept of pervasive gaming and extend it.

McGonigal sees that in pervasive and immersive game situations gamers maximize their play experience by *performing* the role of believing that they are able to permeate the game-reality boundary. In other words, players don't actually believe that real life and games are the same thing, but their enjoyment flows from pretending that the two worlds are able to be intermingled.

McGonigal describes this gameplay as a combination of the "Pinnocchio effect" – the desire for a game to be transformed into real life – and conversely, the desire for everyday life to be transformed into a "real little game." McGonigal focuses on two examples in her paper: the 2001 immersive game known as the Beast, and the Go Game, an ongoing urban superhero game [ibid].

The OutRun project is designed to fail as a device that will seamlessly permeate the boundary of real life and a game. In other words, it is intentionally built with faulty logic at its core: attempting to map a 1986 video game into the real world likely will never result in a seamless transition between game and life. When the game is extended beyond its normal constraints as a videogame, it results in a malfunctioning: confusion, collisions or accidents. However, this is part of the fun of the OutRun project.

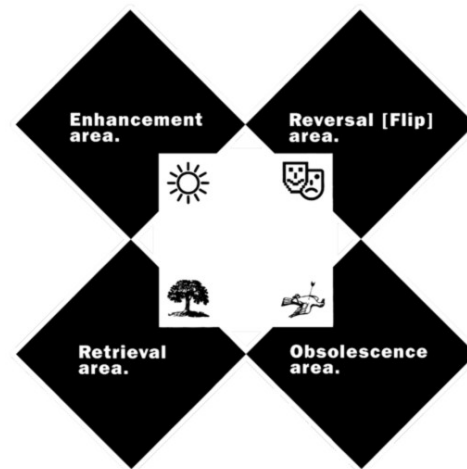


Figure 14. McLuhan's Tetrad diagram, which proposes how communication and representation technologies have the potential to change in four distinct ways: obsolescence, retrieval, enhancement, and reversal.

A useful framework for visualizing and rethinking this dynamic includes Marshall and Eric McLuhan's Laws of Media: The New Science (1989), which proposes that media technologies have the potential to change in four distinct ways [Marshall and Eric McLuhan, Laws of Media: The New Science, 1989]. A poetic four-region model is presented to envision the characteristics of obsolescence, retrieval, enhancement, and reversal (Figure 14). The reversal, in McLuhan's eyes, occurs when something is pushed to its limits.

The main point of relevance to this discussion on pervasive and immersive games is that when mediating technologies are pushed beyond their ordinary limits, they can reverse or flip in their intent or use. The McLuhan tetrad diagram gives a visual graph to think about how game projects may simultaneously amplify, invert, revive, and subsume – they swirl around and don't simply proceed in a straight line. When a simulation is taken beyond its role as a safe fantasy and pushed to envelop and take over reality, it becomes perverted. Problematising a binary virtual/physical view of the world, for example, can be done by taking virtuality to an extreme. By taking a belief or simulation too far, it stops being a comforting diversion and flips into an absurdity, an obsession or a dark dream [Hertz, Un-Simulations, Tricksters and Radical Thought, Blackflash Magazine, 2009]. In the process, beliefs are "perverted" [Rafael Lozano-Hemmer, Perverting Technological Correctness, Leonardo 29:1, 1996].

OutRun does not propose a solution to a problem: it intentionally magnifies and embraces an impossible gap between simulation and reality. It does not see a *line* between game and real life, but rather a pleasurable impossible *area*: an uncanny valley between the familiar and the almost-familiar [Masahiro Mori, 1970]. Exploring this “perverted” and uncanny valley by over-extending simulations is a useful approach for game design. Intentional misuse of technology, incorporating physical pain, questioning cultural concepts of progress, creating intentionally useless devices and bending cultural stereotypes are all useful approaches to embrace in developing the uncanny [Rafael Lozano-Hemmer, *Perverting Technological Correctness*, Leonardo 29:1, 1996].

In this perversion, the game strives to function to break the “natural” rules to highlight and invert common knowledge. To build perverse games is to explore the historical role of a trickster: to disrupt the everyday by highlighting and inverting conventional behavior through humor and paradox. In the process, the contractual magic circle of engagement is extended spatially, temporally, and socially [Markus Montola, *Games and Pervasive Games*, Page 7].

5. ACKNOWLEDGMENTS

Thanks to the development efforts of Walt Scacchi, Professor Jong Weon Lee, Jeremy Bailey, Matt Wong, Erik Olson, Chris Guevara, David Dinh, Matt Shigekawa, Jesse Joseph, and Mike Tang.

The OutRun project has been supported by the Laboratory for Ubiquitous Computing and Interaction at UC Irvine, Gillian Hayes, Christina Lopes, the Arts Computation Engineering Program at UC Irvine, and is currently supported by the Center for Computer Games and Virtual Worlds and the Institute for Software Research at UC Irvine. Support for this research is provided through grants from the National Science Foundation #0808783; Digital Industry Promotion Agency, Global Research and Development Collaboration Center, Daegu, South Korea. No endorsement implied.

6. REFERENCES

- [1] Cycling '74, Max/MSP/Jitter, <http://cycling74.com/>.
- [2] Google, KML – Google Code, <http://code.google.com/apis/kml/>.
- [3] Google SketchUp, <http://www.sketchup.com>.
- [4] Garnet Hertz, *Un-Simulations, Tricksters and Radical Thought*, Blackflash Magazine, 2009.
- [5] Elizabeth Losh, *Hybridizing Learning, Performing Interdisciplinarity: Teaching Digitally in a Posthuman Age*, Digital Arts and Culture 2009.
- [6] Jenna Falk and Gloriana Davenport (2004). "Live Role-Playing Games: Implications for Pervasive Gaming" (PDF). *Entertainment Computing – ICEC 2004. Lecture Notes in Computer Science*. 3166. Springer Berlin / Heidelberg. Page 127.
- [7] F1 ShowCar, <http://f1showcar.com/>.
- [8] Rafael Lozano-Hemmer, *Perverting Technological Correctness*, Leonardo 29:1, 1996.
- [9] Markus Montola, *Games and Pervasive Games*, Page 7.
- [10] Masahiro Mori, 1970.
- [11] Jane McGonigal, "A Real Little Game: The Performance of Belief in Pervasive Play." *Digital Games Research Association (DiGRA) "Level Up" Conference Proceedings*. November 2003.
- [12] Marshall and Eric McLuhan, *Laws of Media: The New Science*, 1989
- [13] The Mirror (newspaper), *SatNav danger revealed: Navigation device blamed for causing 300,000 crashes*, Tanith Carey, 21/07/2008.
- [14] Pac-Manhattan, <http://pacmanhattan.com/about.php>, 2004.
- [15] Tychsen et al., *Live Action Role Playing Games*, page 255.

Industry Data on Young Users in Virtual Worlds Worldwide

Main Source: <http://www.kzero.co.uk/index.php> and others as indicated, circa Feb-Apr 2010.

Note: Not an endorsement nor claim for validity of this data

Note: figures below are expressed in units of millions of users worldwide

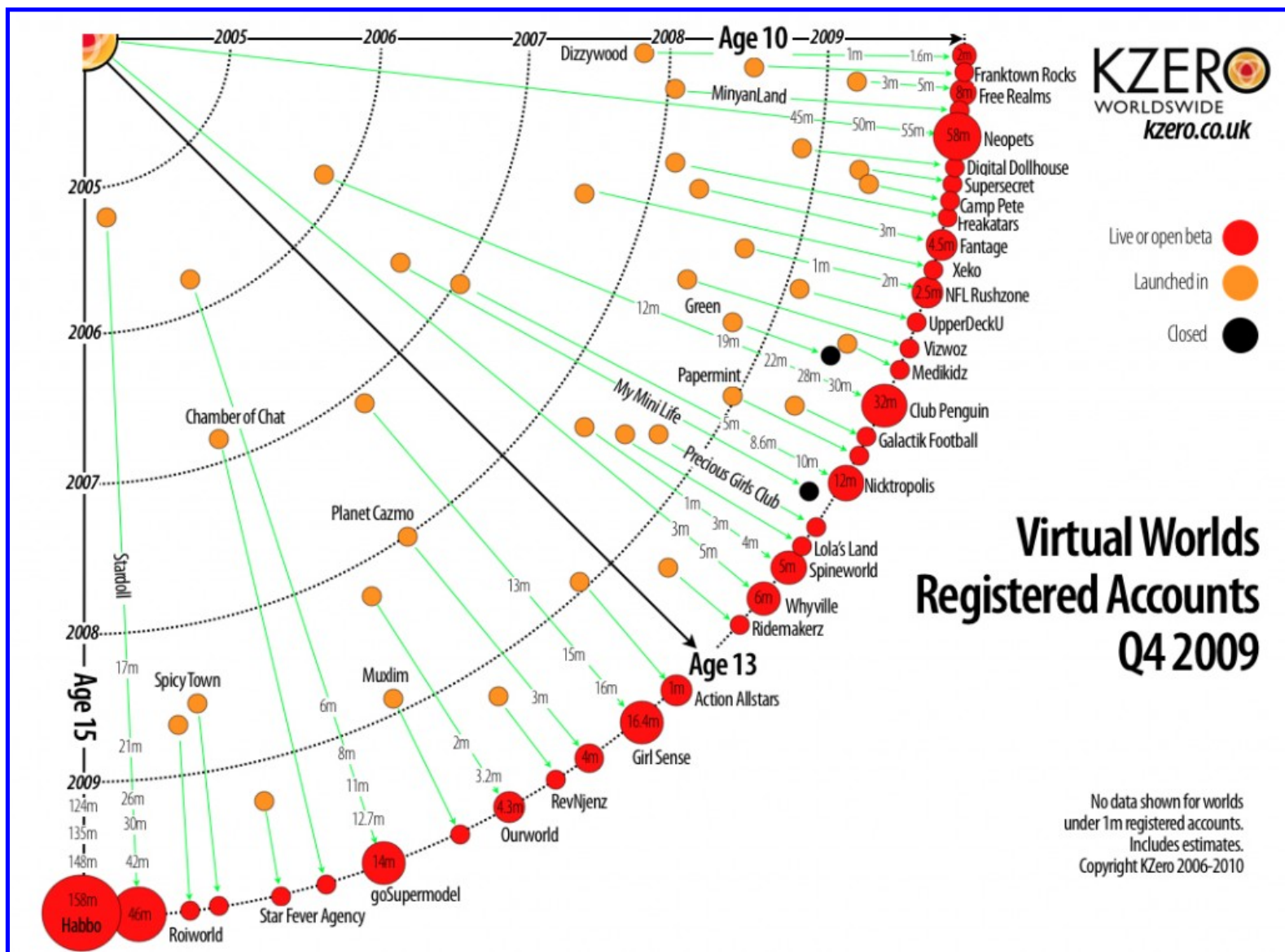
Note: click on the images for larger/detailed view

Note: copyrighted material from KZero and others, displayed here for research discussion purposes only. do not repost.

Q4 2009 Universe chart: Kids and Tweens

As published in their [last post](#), the overall market of total registered users in the virtual worlds sector reached [800m in Q4 2009](#). Here, we delve into the younger segmented of this market, Kids and Tweens.

Virtual worlds with an average age user between five and ten reached a total of 179m in Q4, up 17.8% from 152m. The chart below contains the Universe segment for this age range.

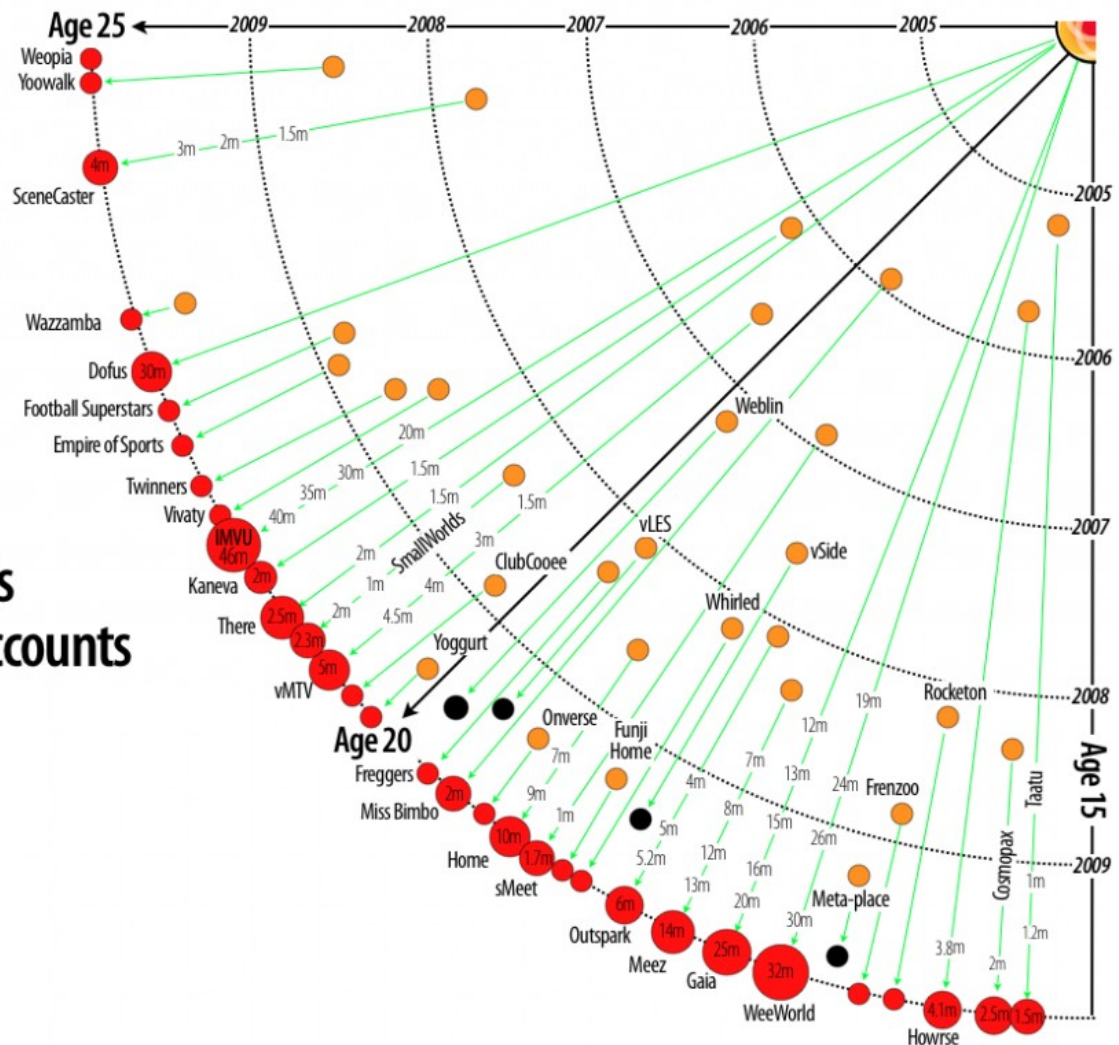


Our last post (assessing [Kids and Tweens](#)) showed the five to 10 year old segment and 10 to 15's grew 17.8% and 6.8% respectively from Q3 to Q4 2009. Here, we cover the older sectors and first up virtual worlds with an average age user from 15 to 25. Here's the Universe chart.



Virtual Worlds Registered Accounts Q4 2009

No data shown for worlds
under 1m registered accounts.
Includes estimates.
Copyright KZero 2006-2010



[IMVU](#) continues to dominate this age range, growing to 46m registered accounts in Q4. One to watch in this range is French VW [Dofus](#), with 30m registered accounts, mainly in France. Of note, IMVU now publishes active users (concurrency) live on their site (with over 100k online at time of writing this post).

On an overall basis, this age range grew from 117m to 193m in Q4 – an increase of 65%. Moving onto the last segment – virtual worlds with an average user age over 25, here's the chart.

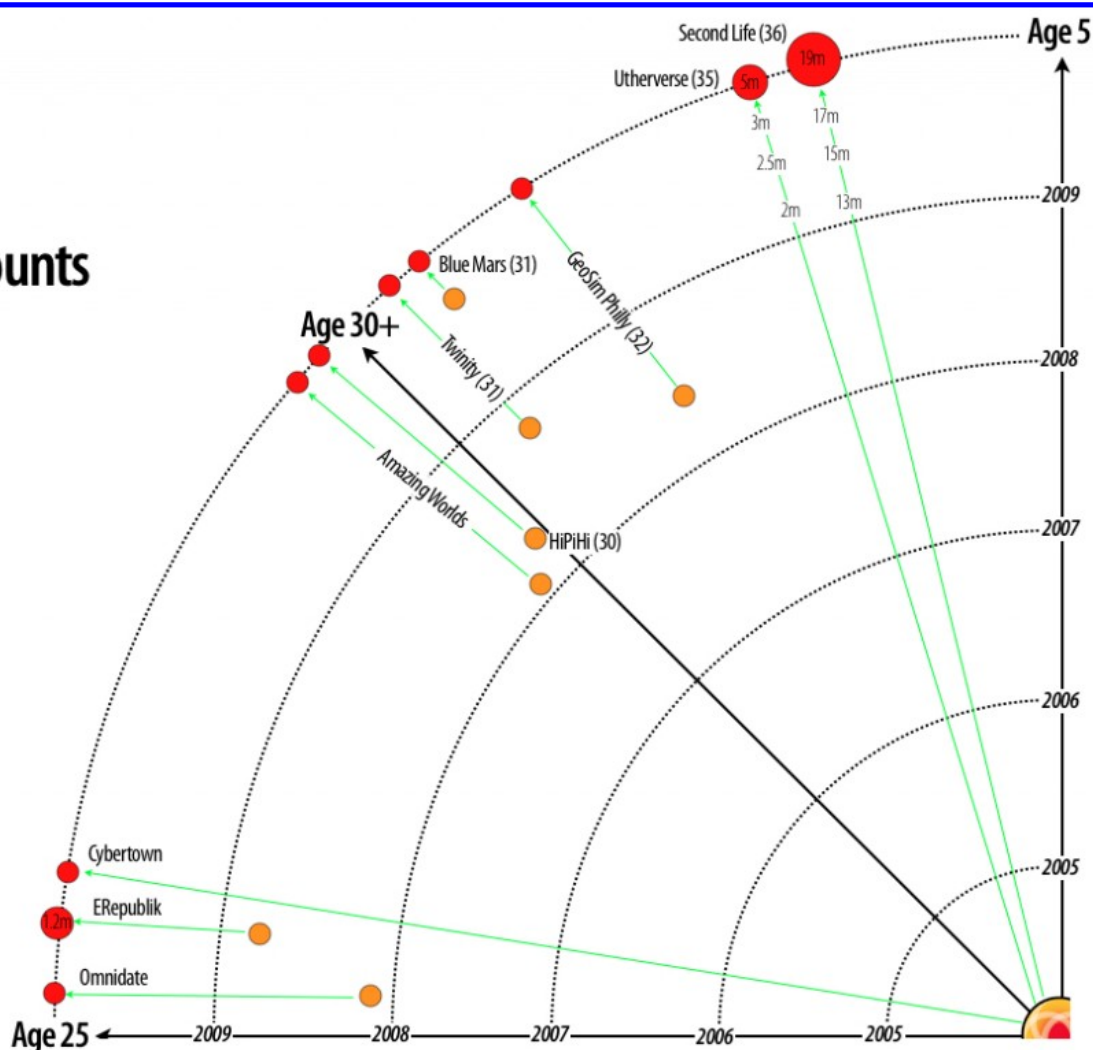


kzero.co.uk

Virtual Worlds Registered Accounts Q4 2009

- Live or open beta
- Launched in
- Closed

No data shown for worlds under 1m registered accounts.
Includes estimates.
Copyright KZero 2006-2010

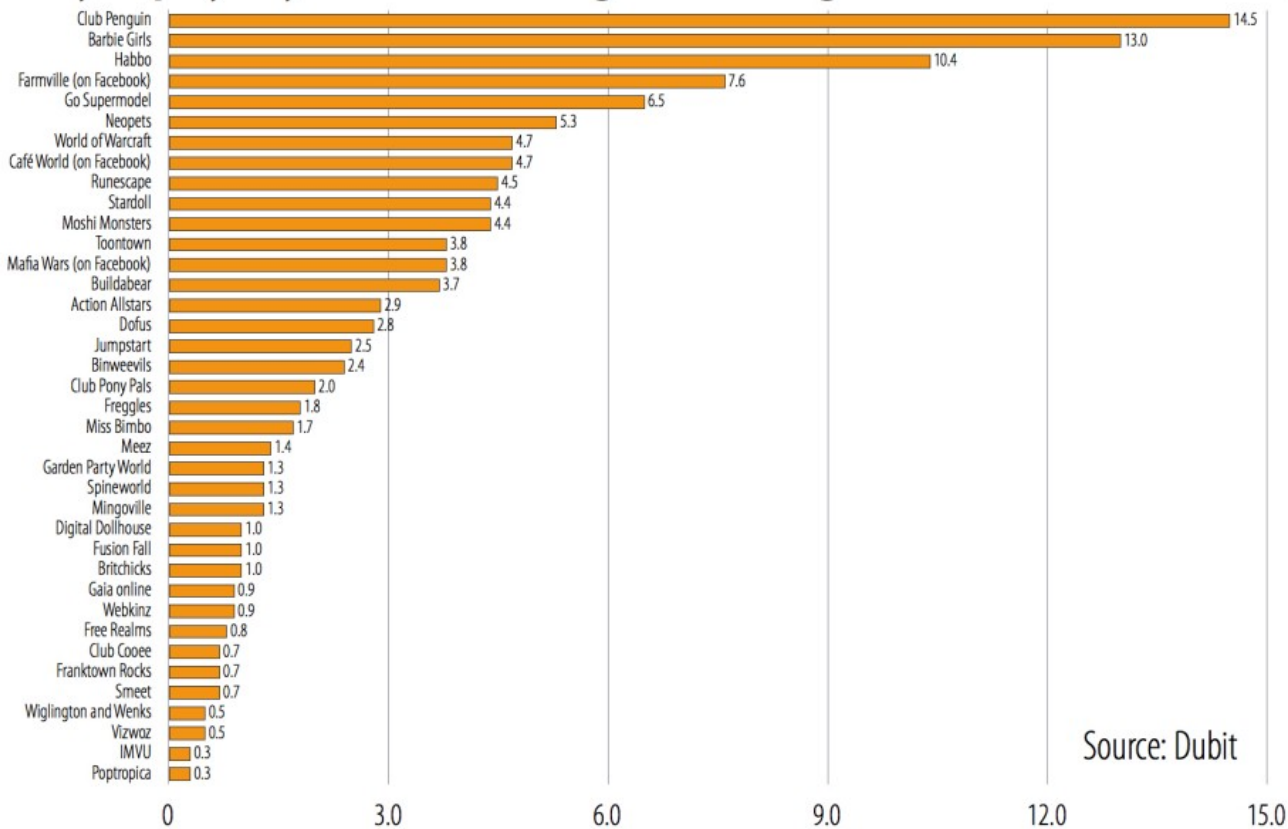


Kids virtual world popularity across the EU -- <http://www.kzero.co.uk/blog/?p=4111>

Here's the master summary of the [Dubit](#) research looking at kids virtual worlds. The countries included in this research were: UK, France, Germany, Holland, Sweden, Finland, Norway and Denmark.

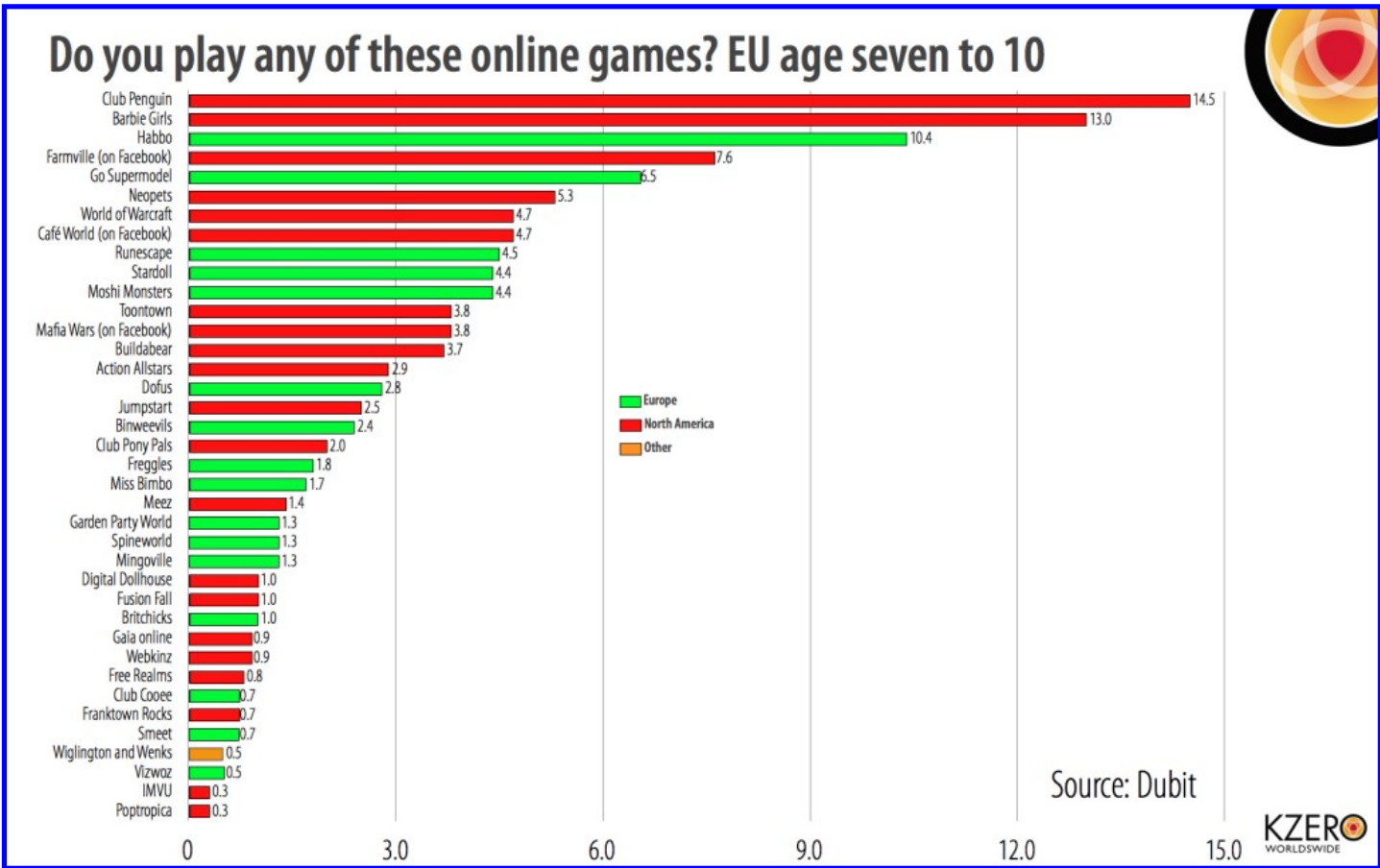
Below is the summary slide ranked by the % of the sample that has played/registered each world.

Do you play any of these online games? EU age seven to 10



Club Penguin comes out top with 14.5% of the sample having played it. Second place goes to Barbie Girls with 13% with Habbo in third. Interestingly GoSupermodel beats Stardoll based on this research, albeit by a couple of % points.

Looking at this EU summary from a company-location perspective...



Excluding Facebook Games and older MMOs, Habbo takes the crown at the top European-based VW, followed by GoSupermodel. Moshi Monsters and Stardoll are next in popularity.

Virtual world registered accounts reach 800m

At the **end of Q4 2009**, total registered accounts in the virtual worlds sector reached 803m. This is a 19.7% (132m) quarter on quarter increase, from 671m registered accounts in Q3. The table below breaks this out by average user age range.

Age Range	Q1	Q2	Q3	Q4
5 to 10	77m	114m	152m	179m
10 to 15	246m	334m	367m	392m
15 to 25	73m	99m	117m	193m
25+	23m	32m	35m	39m
Total	419m	579m	671m	803m

Over the last four quarters on a registered accounts basis the market has almost doubled, going from 419m to 803m.

Looking at this growth by age range (the average user age), the 15 to 25 year old segment demonstrated the highest growth in Q4, representing a 65% increase from 117m to 193m. Strong performance from [IMVU](#) and [Habbo](#) drove this uplift.

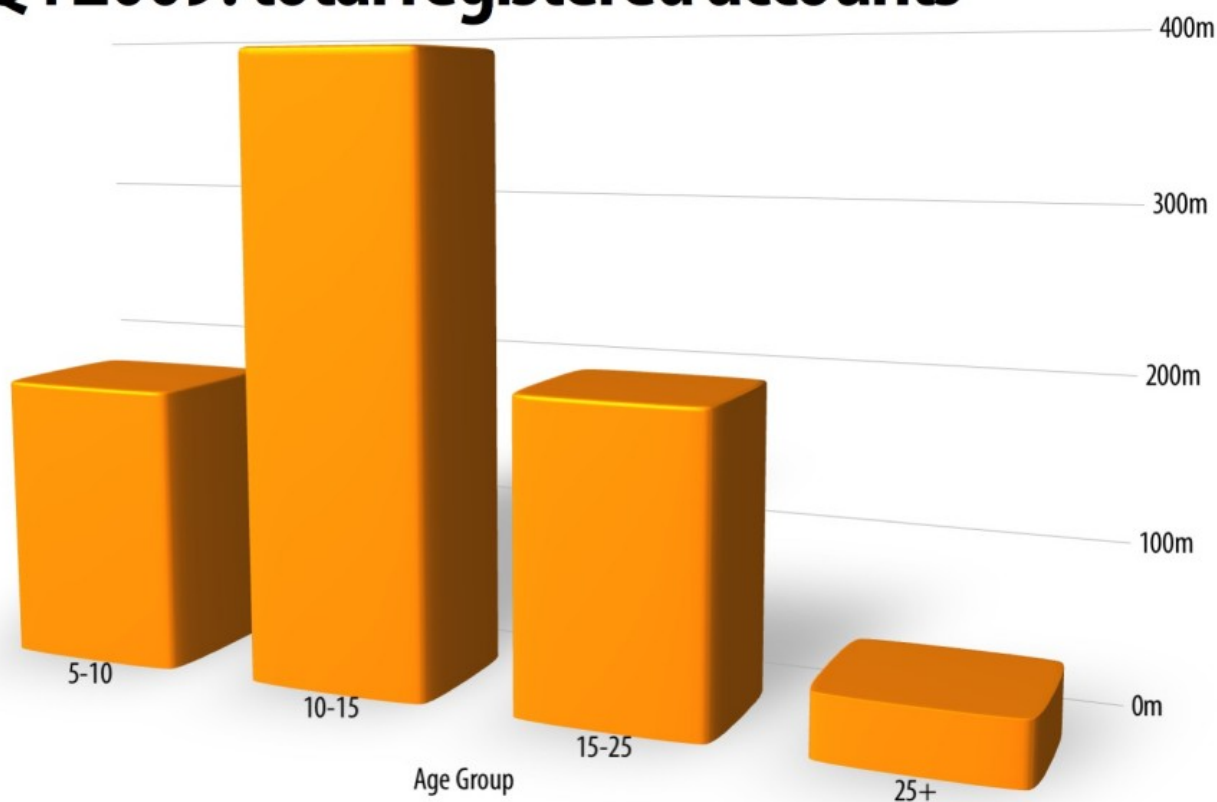
Second highest growth came in the youngest age range (five to 10 year old) and as the just released Radar chart shows, this segment, in particular for virtual worlds catering to education and development is hotting up. Quarter on quarter growth in this segment was 27m, moving from 152m to 179m.

Our Universe and Radar charts have also been updated based on closing Q4 data. Here's a post for the [Universe](#) and another for the [Radar](#).

The full report covering growth in the virtual worlds sector can be ordered [here](#).

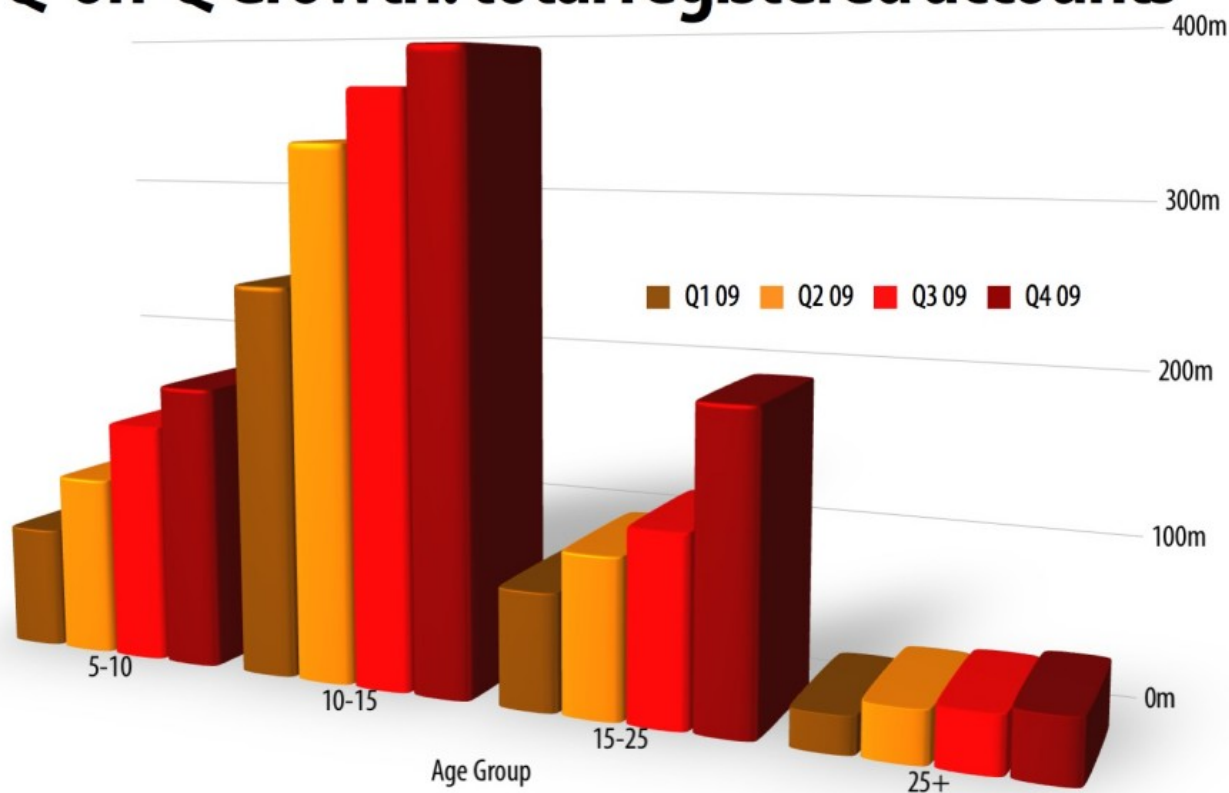
Here's (of course) a few of charts.

Q4 2009: total registered accounts



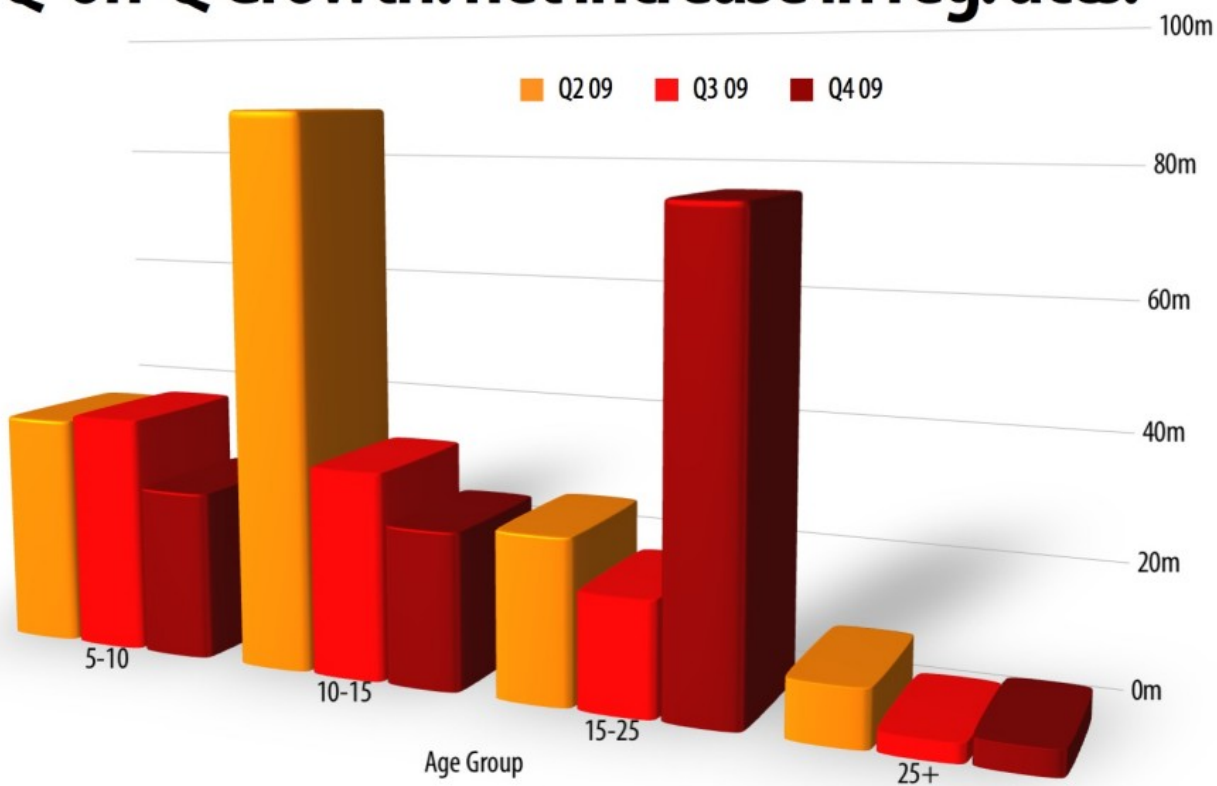
KZERO
WORLDWIDE

Q-on-Q Growth: total registered accounts



KZERO
WORLDWIDE

Q-on-Q Growth: net increase in reg. accs.



Computer Game Mods, Modders, Modding, and the Mod Scene

Walt Scacchi
Institute for Software Research
and
Center for Computer Games and Virtual Worlds
University of California, Irvine 92697-3455 USA
December 2009

Final version to appear in *First Monday*, <http://www.firstmonday.org>, Spring 2010.

Abstract

Computer games have increasingly been the focus of user-led innovations in the form of game mods. This paper examines how different kinds of socio-technical affordances serve to organize the actions of the people who develop and share their game mods. The affordances examined include customization and tailoring mechanisms, software and content copyright licenses, game software infrastructure and development tools, career contingencies and organizational practices of mod teams, and social worlds intersecting the mod scene. Numerous examples will be used to ground this review and highlight how such affordances can organize, facilitate or constrain what can be done. Overall, this study helps to provide a deeper understanding of how a web of associated affordances collectively serve to govern what mods get made, how modding practices emerge and flourish, and how modders and the game industry serve each others' interests, though not always in equivocal terms.

Introduction

Computer game mods are a leading form of user-led innovation in game design and game play experience. But modded games are not standalone systems, as they require the user to have an originally acquired or authorized copy of the unmodded game. Thus, there are questions of not only who creates what and who owns such modified games, but also whether or how the practice of game modding is controlled or governed by external parties to ultimately exploit the efforts of game modders. Conversely, if game mods are co-created by game studio developers and user-modders, then it may also be the case that modders create or enable developers, and thus modders govern some of the actions and choices of developers. In this article, we examine these notions by investigating an array of loosely coupled affordances among people, information resources, computing technologies, and institutional arrangements that are expressed or embodied through game mods, modding practices, modders, and the mod scene.

Affordances refer to situated, interactional properties between objects (tangible or symbolic) and actors that facilitate certain kinds of social interactions within a complex environment. The concept of affordances seeks to characterize aspects of complex work settings that facilitate how people interact though socio-technical systems like computer games. Computer-supported game play environments, when effective, afford new ways and means for collaborative learning [Gee 2008, Hayes and Games 2008, Scacchi, Nideffer and Adams 2008]. However, affordances are neither universal nor ubiquitous, nor are they experienced in the same way in different settings. Accordingly, an affordance that enables or encourages certain kinds of actions in one game mod setting, may inhibit or discourage similar actions in another setting. Modding practices observed for one game may differ from another game, so details and settings do matter. Subsequently, the focus in this article is directed at the interactions that facilitate collaborative activities between game modders who are geographically dispersed but share

access to online artifacts, networked information repositories and communication infrastructures, such as Web pages, Web sites, source code version servers, distributed file servers, and virtual private networks.

Affordances are enablers of collaboration and governance. For example, the online game modding artifacts, circumstances, and boundary objects can make different forms of governance obscure or coordination/control less visible [Star and Strauss 1999] and more remote. However, the information resources, social arrangements, technological system configurations, personal and professional interests that surround game mods, modding, and modders continually emerge as a web of associations or interrelated affordances that stabilize actions and practices [Latour 2003]. Consequently, our interest lies in examining where such associations stabilize and entrench the “social” surround of computer games in different settings, and at different levels of analysis [Kling and Scacchi 1982, Scacchi 2004, Scacchi 2008].

Governance can therefore be perceived through a web of affordances that either push it into a nearly invisible background, or into a visible confrontational foreground, in ways that may be historically situated or transcendent across space, time and circumstance. Thus looking at, into, or through a web of associations requires multiple lenses with different focal properties to discern how affordances widely available across the game community collectively act to govern who mods what, where, when, how and why.

Mods, Modders, Modding, and the Mod Scene

Depending on who you ask or where you look, game mods mean different things to different people. So if we seek to understand how game mods (objects), modding (activities or practices), modders (people working in particular situations), or the mod scene (the surrounding social world), we frame our study as one encompassing multiple affordances that naturally arise at different levels of granularity. Accordingly, we observe activities at levels that span (a) game mod embodiments, (b) game software and content licenses, (c) game modding software infrastructure and development tools, (d) career contingencies and organizational practices of game modders, and (e) social worlds intersecting the mod scene. While no claim is made that such levels are unique or definitive, their role is to help ground and reveal how governance of game mods, modding practices, modders, and mod scene occurs across a society of games.

We thus turn to examine modding at each of these levels, starting with what constitutes a mod.

Customizing, Tailoring, and Remixing Game Embodiments

Understanding what constitutes a game mod is not as simple as it might seem. Said differently, if you in some way alter a game, how its played, or on what it is played, have you created a game mod? Since we prefer not to presume some prior definition of game mods that might privilege the inclusion or discount the exclusion of some mods, modding practices, or modders, we choose to employ a wide-ranging view. So to start, we conceive of game mods as covering customizations, tailorings, and remixes of game embodiments, whether in the form of game content, software, or hardware denoting our space of interest.

As scholars like Dourish [2001] and Suchman [2007] have observed with other computing technologies, understanding game mods starts from observing how players interact with and reconfigure the game embodiments at their disposal. Computer games and game mods take on tangible form that reflect certain kinds of social computing experiences that are mediated by game system platforms [Montfort and Bogost 2009]. Modding is a “Do It Yourself” approach to technology

personalization that can establish both socio-technical and distributed cognitions for how to innovate by resting control over technology design from their producers. Modding is a form of *meta-gaming* — playing games for playing with the game systems. At least five types of game mods can be observed: user interface customization; game conversions; machinima and art mods; game computer customization; and game console hacking. Each enables different kinds of affordances that govern mods, modding practices, and modders.

User interfaces to games embody the practice and experience of interfacing users (game players) to the game system and play experience designed by game developers. Game developers act to constrain and govern what users can do, and what kinds of experience they can realize. Some users in turn seek to achieve some form of competitive advantage during game play by modding the user interface software for their game, when so enabled by game developers, to acquire or reveal additional information that the users believe will help their play performance and experience. User interface add-ons subsequently act as the medium through which game development studios can support mass customization of their game products, as a strategy for increasing the likelihood of product success through user satisfaction [Pine 1992]. Three kinds of user interface customizations can be observed. First and most common, is the player's ability to select, attire or accessorize the characters that embody a player's in-game identity. Second, is for players to customize the color palette and representational framing borders of the their game display within the human-computer interface, much like what can also be done with Web browsers and other end-user software applications.¹ Third, are user interface add-on components that modify the player's in-game information management dashboard that do not necessarily modify game play rules or functions. These add-ons provide additional information about game play that may enhance the game play experience, as well as increasing a player's sense of immersion or omniscience within the game world through sensory or perceptual expansion. This in turn enables awareness of game events not visible in the player's current in-game view [Taylor 2006], as the cognition required for sustained game play become more distributed [Taylor 2009]. Kow and Nardi [2010] describe their study of add-on modding practices for *World of Warcraft* in a cross-cultural setting, which helps reveal that cultural practices in different nations may encourage or not encourage such modding, as a reiteration of norms that tacitly govern social collectivity and conformity, and that eschew radical individuality and innovation.

Game Conversion Mods are perhaps the most common form of game mods.² Most such conversions are partial, in that they add or modify (a) in-game characters including user-controlled character appearance or capabilities, opponent bots, cheat bots, and non-player characters, (b) play objects like weapons, potions, spells, and other resources, (c) play levels,³ zones, terrains, or landscapes, (d) game rules,⁴ or (e) play mechanics.⁵ Some more ambitious modders go as far as to accomplish either (f) total conversions that create entirely new games from existing games of a kind that are not easily determined from the originating game, or (g) parodies that implicitly or explicitly spoof the content or play experience of one or more other games via reproduction and transformation. For example, one of the most widely distributed and played total game conversions is the *Counter-Strike* (CS) mod of the *Half-Life* first-person action game from Valve Software. As the success of the CS mod gave rise to millions of players preferring to play the mod over the original HL game, then other modders began to access the CS mod to further convert in part or full, to the point that Valve Software modified its game development and distribution business model to embrace game modding as part of the game play experience that is available to players who acquire a licensed copy of the HL product family.⁶ Valve has since marketed a number of CS variants that have sold over 10M copies as of 2008, thus denoting the most successful game conversion mod, as well as the most lucrative in terms of subsequent retail sales derived from a game mod.⁷ Other player-modders of *Half-Life* games have also benefited by access to meta-mods, such as *Garry's Mod* of *Half-Life 2*, which has evolved into a modding toolkit⁸ that has generated hundreds of game conversions and inventive game play mechanics. Though the success of a

single game conversion may not foretell a similar success of any subsequent mod (as no game conversion to date has reached anywhere the global success attained by CS), other game conversions represent innovations in game design and re-purposing. For example, the game *Chex Quest* is a conversion of the first-person shooter, *Doom*, into a “non-violent” game that was distributed in the Chex brand of cereal boxes on a CD targeted to young people and gamers.⁹ Another example is found in games converted to serve a purpose other than entertainment, such as the development and use of games for science, technology, and engineering applications. For instance, the *FabLab* game [Scacchi 2010] is a conversion of the *Unreal Tournament* game, from a first-person action shooter to a simulator for training semiconductor manufacturing technicians in diagnosing and treating potentially hazardous materials spills in a cleanroom environment. However, this conversion is not readily anticipated by knowledge of the *Unreal* games or underlying game engine. Finally, another class of game conversions are found in the form of game parodies — games created to reproduce and reframe the game play concepts and mechanics found in one or more popular games. For example, the game *WTF?!* is a 2D, Flash-based, side-scrolling, parody of the 3D *World of Warcraft* game that shares exportable character information from WoW.¹⁰ But *WTF?!* introduces new characters like Sigmund Freud, Karl Marx, and Mary Daly who serve to mod the WoW play experience through commentaries and quest directives that act to direct the player's attention respectively to construction of identity of player game characters, alternatives to capitalistic resource accumulation as game play goal, and dealing with the objectification of the female form within games.

Machinima and Art Mods can be viewed as the product of modding efforts that intend to mod play experience.¹¹ These products employ computer games as their creative media, such that these new media are mobilized for some other purpose (e.g., creating online cinema or interactive art exhibition). Machinima focuses attention to playing and replaying a game for the purpose of story telling, movie making,¹² or retelling of daunting or high efficiency game play experience [Lowood 2008]. Machinima is a form of modding the experience of playing a specific game through a recording of its visual play session history so as to achieve some other ends beyond the enjoyment (or frustration) of game play. These play session histories could then be modded via video editing or remixing with other media (e.g., audio recordings) to better enable cinematic storytelling or creative performance documentation. Art mods also modify the game play experience through manipulation, intervention, appropriation, or other creative transformation of a game's original visual content as it is consumed by users during during a play session. Artists working in the interactive medium of games have explored appropriation and intervention as tactics for using modded games as static, dynamic, or performance art works.¹³ Artists were among the first community of game players to embrace game hacks and patches: modifications to original game software functionality as a strategy for creating new works of art intended for curation and exhibition. The 1999 “Cracking the Maze” art exhibit curated by Anne-Marie Schleiner helped bring these modified games to light through an online exhibit that was structured as a C++ source code program for viewing or interaction in an Emacs-like text editor.¹⁴ Art mods can also incorporate alternative game control devices and user interface non-sequiturs.¹⁵

Custom gaming PCs are another expression of game system modding practices. Here players direct their efforts not to merely accessorizing their game platforms, but to go under the hood and to assemble, reassemble, or otherwise reconfigure their general-purpose personal computer into to one that is specialized for competitive game play or display in a multi-player environment [cf. Suchman 2007]. What results are “hot rod” and “custom car” PCs, computer hardware platforms modded for either maximum speed and performance, or special appearance for case mods [Simon 2007], rather than for mere operation of mainstream software applications or casual game play. Hot rod game PCs often feature high performance and over-clocked CPUs with liquid-cooling devices, custom-timed on-board memory chipsets, multiple GPU cards, and a variety of accelerator cards for in-game physics, networking, game audio, and so forth.¹⁶ These PCs embody a desire to achieve competitive game play

advantage for some, or sheer display of performance or signification of performance potential, much like automobiles are modded by their owners for the purpose of increasing their driving performance on the road, or as might be called for in a street racing situation.¹⁷ Though public practice of such situations with hot rod street cars is often isolated and criminally penalized when excessive, PC platform hot rodding is yet to achieve such distinction, public recognition, consequence, or cinematic embodiment. In contrast, PC case mods, which generally do not modify game functionality or performance, serves to signify a game player's interest or technological projection of self-identity onto their game play platform. Such projection denotes an unabashed choice to display one's enthusiasm, alignment, and commitment to game play as more than just entertainment, but as part of one's personal identity, fetish, cultural experience and life-style preference. Nonetheless, enthusiasts who do so mod their computers, also show off their machines in more public venues like regional LAN parties or game play conferences like QuakeCon.¹⁸ Custom gaming PCs thus represent a strategy for game play enthusiasts to proudly assert their embrace of game culture and the vendors that supply the means to display such embrace, though while being able to pose in stance that signifies independence from the governance telegraphed by corporately designed PCs.

Game console hacking is a practice whose purpose oftentimes seems to be in direct challenge to the authority of game console manufacturers that represent large, global corporate interests. Console hacking, in contrast to custom PC (re)configuration, is often focused not so much on how to improve competitive advantage in multi-player game play, but instead is focused on expanding the range of experiences that users may encounter through a game console [Consalvo 2007, Huang 2003]. For example, Huang's [2003] treatise on how to hack into a Microsoft Xbox console stresses how to unlock the potential of the Xbox as an alternative personal computer that could also support personal productivity applications or services, along with playing proprietary games acquired from Microsoft game vendors. However, Huang's study instructs readers in the practice of "reverse engineering" as a strategy to understand both how a computing platform was designed and how it operates in fine detail, as a basis for developing new innovative modifications or original platform designs, such as installing and running a Linux open source operating system (instead of Microsoft's proprietary closed source offering).¹⁹ While many game developers seek to protect their intellectual property from reverse engineering through end-user license agreements, whose terms attempt to prohibit such action under threat of legal action, reverse engineering is not legally prohibited nor discouraged by the Courts. Consequently, the practice of game console modding is often less focused on enabling players to achieve competitive advantage when playing retail computer games, but instead may encourage those few so inclined in the practice for how to understand and ultimately create computing innovations through reverse engineering²⁰ or other DIY hardware modifications.²¹ Console modding is thus an expression of game players who are willing to fore go the "protections" and quality assurances that console developers provide through product warranties, in order to experience the liberty, skill and knowledge acquisition, as well as potential to innovate, that mastery of reverse engineering affords. So while game console developers and retailers may seek to govern the actions that discourage users from looking inside of the gaming platforms, players who are willing to take responsibility for their actions (and not seek to defraud vendors due to false product failure warranty claims), can enjoy the freedom to learn how their gaming systems work in intimate detail and to potentially learn about hardware innovation with the support of others like-minded.

Software and Content Licenses

Players acquire computer games through retail outlets (online or offline stores). But what they pay for when they "buy" a game is a license to use the game product or product media. This license is commonly expressed as a copyright-based, end-user license agreement (EULA), much like it is for

most other software that consumers acquire. The terms and conditions of such license stipulate what rights and obligations a user of a licensed game copy realizes, as the game software and content is owned by the game's development studio, publisher, or distributor. So here we examine how a game's software and content license(s) can govern mods, modding, and modders.

First, for game conversion mods, it is common practice that the underlying game engine has one set of license terms and conditions to protect original work (e.g., no redistribution), while game mod can have a different set of terms and conditions as a derived work (e.g., redistribution allowed only for a game mod, but not for sale). In this regard, software licenses embody the business model that the game development studio or publisher seeks to embrace, rather than just a set of property rights and constraints. For example, in *Aion*, an MMOG from South Korean game studio NCSoft, no user created mods or add-ons are allowed, so attempting to incorporate such changes would conflict with its EULA and subsequently put such user-modders at risk of losing their access to networked *Aion* multi-player game play. In contrast, *WoW*, also an MMOG, allows for UI customization mods and add-ons only [Kow and Nardi 2010], but no other game conversions, no reverse engineering game engine, and no activity intended to bypass *WoW*'s encryption mechanisms.²² And, in one more variation, for games like *Unreal Tournament*, *Half-Life*, *NeverWinterNights*, *Civilization* and many others, the EULAs encourage modding and the free redistribution of mods without fee to others who must have a licensed game copy,²³ but no reverse engineering or redistribution of the game engine required to run the mods. This restriction in turns helps game companies realize the benefit of increased game sales by players who want to play with known mods, rather than with the unmodded game as sold at retail. Mods thus help improve games sales, revenue, and profits for the game development studio, publisher, and retailer (see note 7 for details).

Second, it is sometimes noted that game software licenses fundamentally differ based on whether they seek to insure copyright or copyleft, where the former provides closure and prevents certain freedoms to act, while the latter insures openness and certain freedoms — freedom of choice, freedom of expression, and freedom to modify and redistribute, plus obligation to propagate these freedoms. So it is sometimes stated that game conversion mods can be viewed and treated as free software. But such a position is an overstatement of fact in that free/open source software that is protected with a GPL license covers the source code and embedded documentation, but not game assets, contents, play mechanics, or rules of play, which can also be subject to protection through copyright, trademark, patent, trade secret, and restricted use. So as a growing number of games allow for modding and redistribution of mods, then we should expect to see a growth and diversification of the number of licenses, or EULA terms and conditions, that will constrain the rights and obligations that will apply to games, game engines, game assets, and game modders. Early signs of this have already begun to appear with heterogeneous licensed game development platforms like *Unity3D*, which itself is subject to more than a dozen different intellectual property (IP) licenses [Alspaugh, Asuncion, Scacchi 2009].

Similarly, much like some open source software, such as the Mozilla Firefox Web browser, open game engines may become subject to dual-mode, tri-mode, or multi-mode copyright licenses.²⁴ Finally, “open gaming licenses” have begun to appear which allow/restrict game configurations that include open source software along with proprietary data assets/artwork, game play rules, or game product identity.²⁵ In sum, game software and content licenses act to govern what mods can or cannot be made, by whom, and with what subsequent rights and obligations. But it is also clear that such governance is only effective where considered legitimate, as there are those who will ignore declared IP restrictions or act to crack open games to mod them in some way, and to share their results with others. Thus we should recognize that there is also resistance to governance, in the form of those who choose to ignore the license terms and conditions that come with closed, restricted games.

Game Software Infrastructure and Development Tools

Games are most often modded with tools that provide access to an internal, unencrypted representation of the game software²⁶ or game platform. While it might seem the case that game vendors would seek to discourage users from acquiring such tools, we observe a widespread contrary pattern. Game system developers are increasingly offering software tools for modifying the games they create or distribute, as a way to increase game sales and market share. SDKs and other modding tools provided to users by game development studios represent a contemporary business strategy for engaging users to help lead product innovation from outside the studio [von Hippel and Katz 2002, Jepperson 2005]. Once id Software and Epic Games, maker of the *Unreal* series of game, started to provide game play enthusiasts (who were also literate in computer programming) with software tools that would allow users to edit game content, play mechanics, rules, or other functionality, other competing game development studios were pressured to make similar offerings or face a possible competitive disadvantage in the marketplace. However, these tools do not provide access to the underlying source code that embodies the game engine—a large software program infrastructure that coordinates computer graphics, user interface controls, networking, game audio, access to middleware libraries for game physics, and so forth.

A tool support platform allows a game development studio to offer game products and services that users can mod. At the same time, studios seek to control access to the core IP that enables game play and modification. Accordingly, we can observe different strategies for how game development studios configure and support game modding platforms. For example, *World of Warcraft* from Blizzard Entertainment, provides a a UI Customization tool and a governance policy²⁷ and that enable WoW add-ons that reconfigure a WoW player's user interface dashboard, which in turn may provide an enhanced game play experience for technically accomplished players. But these add-ons do not modify or convert the WoW game into something entirely different, since Blizzard seeks to insure and control that users with or without add-ons have access to the same WoW in-game play setting and game play mechanics. In contrast, BioWare offers its *NeverWinterNights* game series with its Aurora Toolset and module construction toolkit,²⁸ FAQs, online tutorials, and in-game character profiles to help would be user-made content creators to modify NWN game play levels and characters. But creating new characters for such mods requires additional tools (e.g., modeling software like Autodesk 3D Studio) and corresponding specialist skills. Similarly, Epic Games has recently announced that it is distributing a suite of a dozen or so game development tools as the *Unreal Development Kit*,²⁹ which is said to be the same SDK that Epic previously marketed and distributed only to other game development studios that bought its commercial game software development licenses. However, the complexity and capabilities of such a tool suite mean that any one, or better said, any game development or modding team, can now access the tools to build commercial quality games, though mastering these tools appears to be a significant undertaking likely to be only of interest to highly committed, would-be game developers who are self-supported or self-organized.

Subsequently, we observe a range of corporate governance and control schemes that vary across game development firms, from (a) highly focused and limited at Blizzard for WoW, to (b) supported mediation in NWN by Bioware, to (c) do whatever you can if you have the skill, resources, team, and other resources to do so and sell your results, if you pay for such right with Epic Games. But all these schemes still stipulate control over the underlying game engine, such that it is out of reach to game players, as a way to protect key IP resources of the game studios.

In contrast to game modding platforms provided by game development studios, there are also alternatives provided by the end-user community. One approach can be seen with facilities provided in *Garry's Mod* mod-making package³⁰ that you can use to construct a variety of fanciful contraptions, to create comic books, to program game conversions, or to produce other kinds of user created content.

But this package requires that you own a licensed game like *Counter-Strike: Source*, *Half-Life 2* or *Day of Defeat: Source*, all from Valve Software. A different approach to end-user game development platforms can be found arising from free/open source software games and game engines. As noted above, the *Doom* and *Quake* games and game engines are released as free software that are subject to the GPL,³¹ once they were seen by id Software as having reached the end of their retail product cycle. Hundreds of games/engines have been developed and released for download starting from the free/open source software that was the platform of the original games. However, the content assets for many of these games (e.g., in-game artwork) are not covered by the GPL, and so user-developers must still acquire a licensed copy of the original game if its content is to be reused in some way. Nonetheless, some variants of the user-created GPL'd games now feature their own content that is limited/protected by Creative Commons licenses.

Career Contingencies and Organizational Practices of Modders Alone and Together

Who gets “paid” to make games when comparing developers versus modders? Developers who work for established game studios get paid a monetary salary and sometimes stock options (*financial capital*) to make game content, software, or play devices. Modders in general do not get a paid to make games, unless they are unusually successful by winning an open modding competition [Sotamaa 2007] or landing an angel/venture capital investment. However, modders can earn or acquire other kinds of career-oriented resources of professional or occupational value. These resources include deep game development knowledge, skills for how and when to apply such knowledge, status, community recognition and reputation, and other forms of *social capital* for the gifts of labor, creativity, expertise, and collaboration efforts they contribute [Portes 1998]. In this regard, both employed game developers and independent game modders can build on the efforts of one another in ways that are asymmetric and non-equivalent. Nonetheless, such an relationship is valuable to those who are committed to a career in game development inside or outside of the computer game industry. This is similar to what has been observed in a number of studies of open source software development projects, where in some cases, corporations that sponsor an open source project look to recruit and hire project contributors who make substantial and highly regarded software contributions to a project [Scacchi 2007].³² Furthermore, it has also been observed that developers who have a track record of success in open source software projects often get paid a higher monetary salary compared to their workmates who lack comparable experience [Hann, et al. 2002]. So it may be the case that game modders have the potential to invest in themselves as a way to acquire professional identity, accomplishments, and compensation if they choose to direct their career trajectory towards becoming a game developer in a studio. However this remains an open question for further study to substantiate, refine or refute. In the interim, what kind of investment is required of a game modder to enable such career contingencies?

To begin, why can't you get a job working for a game company if you are a software developer or artist? In simple terms, the computer game industry often looks to hire experienced game developers, rather than just game players or fans. Commercial game development is a risky venture, as most retail games often fail to recoup their development costs (as do many commercial software development efforts). Failure to produce a profitable game is all too familiar of an outcome, even for established game studios with skilled developers. Adding new staff with little or no prior game development experience implies the need for training and socialization, which is time consuming and may be unproductive. Game studios tend to hire developers who have some sort of record of game development. So without prior industry experience, starting as an independent game modder is the easiest way to determine whether one has the skill, disposition, and commitment to make games, even without a job to do so. Consequently, game modders must start and follow a self-motivated path.³³

The most direct way to become a game modder is through self-tutoring and self-organizing practices. Modding is a form of learning – learning how to mod, learning to be a game developer, learning to become a game content/software developer, learning computer game science outside or inside an academic setting, and more [El-Nasr and Smith 2006, Hayes and Games 2008, Scacchi 2004]. Modding is also a practice for learning how to work with others, especially on large, complex games/mods. Many would-be modders seek out others who have gone before them, much like game players at lower game play levels sometimes seek advice from players at higher levels for how best to proceed. In acting to become a modder or game developer, it may be reasonable to search out others like minded by joining a modding team, learning modding practices and mod team roles, and why you can't be in charge until you are recognized as being capable of leading.³⁴ Mod team efforts may also self organize around emergent leaders or “want to be” (W.T.B.) game component development leaders, as highlighted in the *Planeshift* open source MMOG development/modding project.³⁵

Modding is also a viable way to get a job in the game development industry. Epic Games for many years has encouraged would-be game developers to download its game editing tools (*UnrealEd*, the early precursor into what is now the *UDK* described earlier), start modding an *Unreal* game, and then perhaps get hired by a game development studio as a experienced game developer.³⁶ Similarly, to stand behind its advocacy of such a career pathway into a game development studio, Epic Games has also sponsored a series of mod-making contests, most recently under the name of “Make Something Unreal” that provides sizable cash prizes (with co-funding from Intel Corporation), or a full commercial license for *UDK*, so that a game conversion mod can be marketed and sold. Such an outcome essentially provides the winning mod team with venture seed capital, but without the venture capitalist's ownership of the new venture. Making game conversion mods may also open pathways for modders from a small game studio with limited resources to move upward into a large game studio with ample resources and multiple game development projects.³⁷ However, some scholars [Kücklich 2005, Somataa 2007] are skeptical about the value of modding, which they see as “playbour” (a coercion of leisurely play into work) or a commodification of leisure time, and subsequently conclude that modding work primarily benefits game companies by exploiting the productive efforts of game modders. However, if game modders are acting out of self-interest to establish a career path into jobs at game companies, and if they get to create games conversions that open new career opportunities in other industries, then it would seem that modders use modding as a way to exploit game development studio's desire to hire skilled game developers. Consequently, our view going forward may now need to reflect that modding as work can be productive to both independent mod makers and commercial game studios, but in different ways that may further reinforce and reproduce one another.

So our position is the marketplace of career contingencies for game developers and game modders governs through an invisible hand of individual, team, and corporate self-organizing interests to help select and guide who will be able to work for paid salary or who will accumulate skill, experience, and social capital as a game or game mod developer.

Social Worlds Intersecting the Mod Scene

Computer gaming is such a large-scale, globally diverse social endeavor that involves hundreds of millions of people as to merit its conception as a computer-centered social movement otherwise known as a *computerization movement* [Elliott and Kramer 2008, Scacchi 2008]. As a computerization movement, it can be employed as an analytical lens at the societal or social world level: a collection of transcendent activities, actors in different/multiple roles, and technical system configurations that are delimited by shared beliefs/ideologies, technical frames for explaining social action and system designs, and structural interests of organizations that contend for control over growth, resource accumulation, and sustainability. They can also be viewed in terms of what other social worlds

(communities or “scenes”) they intersect, which is the point of departure here. Let's consider how the computer game mod scene intersects with the Warez scene, and with the world of free/open source software development.³⁸

The Warez scene³⁹ focuses attention to activities that endeavor to modify protected software and other digital media (recorded music, movies, console circuit designs) so as to make them widely available for unfettered access and consumption. Games are one of the most commonly modified types of software that are transformed in the Warez scene into “pirated games” or game warez that are “illegally downloaded.” The Warez scene in a sense is focused on engaging a kind of meta-game that involves modding game software products. Game piracy has thus become recognized as a collective, decentralized and placeless endeavor (i.e., not a physical organization) that relies on torrent servers as its underground distribution venue for game warez. Game development studios, publishers, and some retailers all frequently lament their belief that game piracy is a kind of theft that is reducing their game sales and profits, as well as circumventing extant Digital Rights Management mechanisms [e.g., Greenburg and Irwin 2008]. As recent surveys of torrent-based downloads reveals, in 2008 the top 10 pirated games represented about 9M downloads, while in 2009 the top 5 pirated games represent more than 13M downloads, suggesting a substantial growth in interest in and access to such modded game products.⁴⁰ And these figures do not account for sharing or exchange of game warez on private peer-to-peer networks or darknets. Finally, it seems that in parts of the physical world like India and throughout Asia, game consoles can be purchased directly from retailers already modded with modchips that allow them to run game warez, including games not available in their national retail markets. This is not to impune some negative social connotation on other cultures, but instead to point out that not all national cultures share Western values or beliefs about what can be shared or what piracy means, and under what conditions or corporate institutions may be affected.

In the case of game warez and modded game consoles, it may be the situation that the developers and distributors of such products still benefit from pirated warez, if they can substantiate that such piracy is theft of products with economic value. Such theft can be treated as a legitimate business deduction when determining tax payments to national tax ministries. That is, piracy reduces the amount of profits that are subject to tax, since established losses due to pirated products (e.g., number of pirated games downloaded from torrents) may be tax deductible. Eliminating all game piracy might then actually work against the financial interests of game studios or publishers, if the people who download do not otherwise act to buy games that would now be unavailable to them. Consequently, downloading pirated games from torrents may just represent another channel for distributing products and realizing benefits, with certain costs.

The Warez scene appears increasingly effective at cracking closed games and distributing modded games (or if you prefer, pirated games) and game platforms compared to the capabilities of game studios, publishers, and console manufacturers. So is such accomplishment merely a sign of growing resistance on the part of game consumers, or does it represent an emerging transformation of the marketplace for how games and mods, modding practices, and modders are developed, distributed, played, remixed, reproduced and re-purposed?

Game mods, modding practices, and modders are in many ways quite similar to their counterparts in the world of free/open source software development. Modding is to games, like FOSS development is to software — they are increasingly becoming a part of mainstream technology development culture and practice. Modders are players of the games they construct, just like FOSS developers are also users of the systems they develop. There is no systematic distinction between developers and users in these communities, other than there are users/players that may contribute little beyond their usage, word of mouth they share with others, and their demand for more such systems. At FOSS portals like SourceForge.com, which in December 2009 indicates more than 380K projects are registered in its

repository, the domain of “games” appears as the third most popular project category with over 38K projects, after the domains of “software development” and “Internet.” Of these game-focused FOSS projects which develop either FOSS-based games, game engines, or game tools/SDKs, all of the top 50 have logged more than 1M downloads. So the intersection of games and FOSS covers a substantial social plane, as both modding and FOSS development are participatory, user-led modes of system development, and both rely on continual replenishment of new participants joining and migrating through project efforts, as well as new additions or modifications of content, functionality and end-user experience [Nieborg 2005, Scacchi 2004, Scacchi 2007, Scacchi 2008]. Mods and FOSS development projects are in many ways experiments to prototype alternative visions of what innovative systems might be in the near future,⁴¹ and so both are widely embraced and practiced primarily as a means for learning about new technologies, new system capabilities, new working relationships with potentially unfamiliar teammates from other cultures, and more [cf. Scacchi 2007].

Overall, game mods, game warez, and FOSS can be recognized as expressive forms of participatory culture. Though as we have found above, there are different modes for governing such participation depending on the affordances at hand. But in each situation, governance realizes an emerging transformation the marketplace of ideas and the means of production from centralized authority with corporate enterprises, to decentralized commons-based peer production [Benkler 2006]. Furthermore, sometimes such production may seek its own non-profit foundation or for-profit incorporation, or it may eschew any established organizational form in order to flourish in an alternative market that may not yet be recognized, or one that is being pursued in hope of preventing its move into a position of dominance.

Discussion

One way for viewing and understanding game mods is through the lens of ownership. Ownership influences or controls the allocation of game modding resources, markets or venues where game mods are transacted, or other decisions pertaining to which game mods are created, shared, played, and remixed/remodded by who, when, and where. Such a perspective centralizes governance with those who control ownership of property (games, game assets, engines, and SDKs) and the means of game mod production. Alternatively, a decentralized view of governance draws attention to intrinsic governance activities, patterns, and practices, where ownership is grounded in individual, small group, or project team action and resources. Based on the preceding examination of game mod embodiments, game software licenses, modding infrastructure and tools, career opportunities, and social worlds intersecting game modding, it seems that it is not the case that one of these two perspectives is more spot on compared to the other, as conditions and circumstances that situate game mods, modding, modders, and the mod scene are more heterogeneous than homogeneous, and sometimes less hegemonic than previous studies of game mods have implied. Instead, it may be better to examine the balances accomplished, remade, unmade, or actively prevented among a web of affordances that situate game mods, modding, modders, and mod scenes.

The affordances described and analyzed in this article all point to a web of associations that encourage, enact, exploit, or inhibit how mods, modding, modders, and mod scenes are governed. Governing this mod squad through the five types of affordances examined means that is much less likely that they are one-sided, predominantly in favor of either established game companies or independent self-organizing modders. Instead, what requires a better understanding is how modding as a cultural practice continually shifts how artifacts, embodiments, technical configurations, and arrangement of participants on all sides govern, value, and reproduce what can be accomplished through game mods.

Conclusions

In the study here, we started with the quest to examine how the mod squad constituted from game mods, modding practices, and modders are governed, and to what ends. Affordances of five different types serves as guideposts at different levels of analysis that helped to navigate the path forward. These affordances center about game mod embodiments, game licenses, game modding tools and infrastructure, career contingencies of game modders as would-be game developers, and social worlds that impinge on the game modding culture. Along the way, we examined, employed, or referenced the kinds of analytical constructs or results that others have previously employed in their studies of game mods.

We have seen that there are no single category of object or thing that uniquely denotes what a game mod is. Instead, mods can appear as game user interface add-ons, game conversions of various kinds and complexity, machinima and interactive art, custom gaming PCs, or hacked gaming consoles. Viewing and locating game mods within such a technological ecosystem provides a better perspective for understanding what mods are or can be, who makes them, how and why. Mods or mod embodiments are therefore actors that participate in governing the mod squad.

We have seen that game software licenses (and by implication, game platform and console hardware designs) embody the business models of game development studios. Further, game licenses are increasingly multi-modal and heterogeneous, thus challenging nearly all to determine who can mod what with what. This suggests that such licenses will become a fertile ground for debate regarding what kinds of mods can be made, what modding practices and tools will be provided or tolerated, and who can mod what. Such debates over licenses as a mode of governance and resistance may increasingly spill over into the Courts as well.

Game modding tools, SDKs, modchips, and impinging open source software alternatives for content creation or game engine transformation are collectively an emerging socio-technical media. And as new media, they serve stimulate, enact, or communicate continuing innovation, growth, and diversification by game development studios, game modders and others. The technologies of game modding thus help govern what kinds of mods and modders will emerge, flourish, or wither over time, as well as how the game industry and modding communities can cultivate and grow new markets for new game products.

Game modders are sometimes viewed as leisure laborers whose playful modding actions primarily contribute to the growth of economic rents collected by game studios. However, modders can also be viewed as independent and self-organizing actors who are acquiring the means for producing their own games or game conversions through self-serving investment in skill and knowledge acquisition, time, effort, and socialization with others like-minded. Modders can sometimes become entrepreneurial, and thus transform themselves from game consumers into producers. Consequently, the career contingencies of game modders cannot be simply assumed to be fully covered by some pre-determined set of producer-consumer arrangements or outcomes. Modders make mods through modding practices and technical means, and this in turn helps to produce more modders. Modding careers, contingencies, and pathways/trajectories thus help govern who becomes a modder and to what end.

The social world of game modding provides a system of beliefs, values, and norms that mod makers and users assimilate and reproduce in online modding forums and Web sites. This culture of game mods is enabled and embraced in the social arrangements and technical system configurations that frame what it means to be a game modder, to engage in game modding, and to make and play with a game mod. However, the world of game modding is not an island onto itself. Instead, like all social worlds, it is one that intersects some other worlds, and where and how these intersections occur is a locus of collective social movement, resource redistribution, market transformation and institutional conflict. Games are being modded by actors in the Warez scene in ways that are transforming who get

access to acquire and play what games where. This outcomes of this might be lamented as piracy or recognized as tax shelter on accumulated economic rents. Game modding is also influenced by, and influencing, the world of free/open source software development and its diverse software development projects and practices. Thus where, how, when, with whom, and with what the world of game modding intersects other social worlds that impinge on its boundaries will govern and frame why such intersections will arise, and to what end.

About the author

Walt Scacchi is a senior research scientist and research faculty member in the Institute for Software Research, and also Director of Research at the Center for Computer Games and Virtual Worlds, both at University of California, Irvine. He received a Ph.D. in Information and Computer Science at UC Irvine in 1981. From 1981-1998, he was a professor at the University of Southern California. Dr. Scacchi returned to UC Irvine in 1999. His research interests include open source software development, computer game culture and technology, virtual worlds for modeling and simulating complex engineering and business processes, developing decentralized heterogeneous information systems, software acquisition, and organizational analysis of system development projects. Dr. Scacchi is an active researcher with more than 150 research publications, and has directed more than 50 externally funded research projects. He also has had numerous consulting and visiting scientist positions with more than 25 firms or institutes, including four start-up ventures. His recent research activities and publications can be found at <http://www.ics.uci.edu/~wscacchi>.

Acknowledgements

Support for this research is supported through grants from the National Science Foundation #0534771 and #0808783, and also the Digital Industry Promotion Agency, Global Research and Development Collaboration Center, Daegu, South Korea. No endorsement implied.

Notes

- 1 For example, the Firefox Web browser provides “themes” and “personas” as a way to customize the appearance of a browser's user interface. See <http://support.mozilla.com/en-US/kb/Using+themes+with+Firefox?bl=n&s=themes>, accessed 30 March 2010.
- 2 Much, but not all, of the legacy of game conversion mods originates from the practices of the game software development studio, id Software, with legendary game developers John Carmack and John Romero. They tired of responding to game players requests for additional game features in their existing game products like *Wolfenstein 3-D* and *Doom*, while they sought to focus their attention to new games in development. Subsequently, they chose to hand over the source code to the *Doom* game to their end-user community, and then make the resulting source code subject to free software licensing rights, while still obligating user-modders to acquire the original retail version of the game to access in-game content assets. Many eager users embraced this open sharing of the *Doom* game engine source code, as it could reveal to a *Doom* player who was also a software developer, how the game engine was designed and coded, and thus how to modify or reproduce such an engine, or to build a new one, based on knowledge of the source code and skill in software development. The details of this story are more comprehensively described elsewhere, such as in *Masters of Doom* book by Kushner [2003], and also in studies by Au [2002], Kücklich [2005], and Morris [2003] among others.
- 3 For example, the game *Little Big Planet*, which features game level creation as one of its core game play mechanics, has realized the creation and upload/sharing of 1M levels within its first nine months of release, and now offers more than 1.3M user-rated levels. See http://www.littlebigplanet.com/en-us/game_guide/ps3/playing/community_levels, accessed 30 March 2010.
- 4 Nomic games (<http://www.nomic.net/>, accessed 30 March 2010) are those where changing the rules of a game is move, such that rule changing, generally with the agreement or vote of the game's players, becomes a play mechanic.
- 5 A popular Web portal where modders share and discuss their mods with modding fans who download, play, and rate the mods, can be found at <http://www.moddb.com>. These fans in turn may be just mod players, or they may be other modders who seek to learn about what others have accomplished, with whom to consult for advice, or with whom to try to join with in developing more ambitious game conversion mods. Mod fans or users also may consult <http://www.usercreated.org> for news and reviews of current mods, accessed 30 March 2010.
- 6 As with the previous note 2 on *Doom*, the history of CS is described elsewhere in many places, from Wikipedia <http://en.wikipedia.org/wiki/Counter-Strike>, to various game fan sites like <http://www.cstrike-planet.com/>, to game modding sites like Planet Counter-Strike at <http://planethalflife.gamespy.com/cs/> and others, accessed 30 March 2010.
- 7 See http://www.gamasutra.com/php-bin/news_index.php?story=21319 for disclosed sales figures from Valve Software, accessed 30 March 2010.
- 8 Game modding toolkits are also identified as “software development toolkits” (SDKs). However, they are not general purpose software development environments, but instead are specific to a game or family of games that share a common game engine. Software engineers designate such toolkits as domain-specific SDKs that center about the use of a domain-specific programming language, like *UnrealScript* for the *Unreal* game engine.
- 9 See http://chexquest.wikia.com/wiki/Chex_Quest and also <http://www.youtube.com/watch?v=C1A6dznZPM>, accessed 30 March 2010.
- 10 See <http://aoedipus.net/> for access to the *WTF?! game*, and also to <http://www.nideffer.net/promo/proj/wtf.html> for an overview. The !SDK also supports subsequent modding of *WTF?* -- see <http://nideffer.net/promo/proj/wtfe.html>, accessed 30 March 2010.
- 11 See <http://www.machinima.com> and <http://en.wikipedia.org/wiki/Machinima> for machinima, while for art mods see <http://www.gamescenes.org/>, http://www.selectparks.net/modules.php?name=News&new_topic=5, and http://en.wikipedia.org/wiki/Video_game_art for links to other descriptions and source examples, accessed 30 March 2010.
- 12 The game, *The Movies*, from Lionhead Studios, provides a game play experience that explicitly focuses on tools and techniques for movie-making, as well as enabling creation of parodies about movie-making studios and stars. Details at <http://www.lionhead.com/TheMovies/MakingMovies.aspx> (accessed 30 March 2010) encourage game players to make their own movies and post them on the Internet.

- 13 *Velvet-Strike* is an example of an interventionist art mod that alters the CS mod of *Half-Life* by artists Anne-Marie Schleiner, Joan Leandre and Brody Condon in 2002, so as to allow for inclusion of anti-war graffiti within the tactical small-group battleground of a CS game level/world. See <http://www.opensorcery.net/velvet-strike/>, accessed 30 March 2010.
- 14 See <http://switch.sjsu.edu/CrackingtheMaze/> and also <http://switch.sjsu.edu/web/v5n2/index2.html>, accessed 31 March 2010.
- 15 See art works by artists like Brody Condon <http://www.tmpspace.com/>, Robert Nideffer <http://www.nideffer.net>, and Eddo Stern <http://www.eddostern.com/>, as well as others found at net art portals like <http://www.selectparks.net>, accessed 30 March 2010.
- 16 Print-based and online periodicals like *MaximumPC* (<http://www.maximumpc.com/>, accessed 30 March 2010) and *CustomPC* (<http://www.custompc.co.uk/>, accessed 30 March 2010) focus attention and provide illustrated guidance for how to modify PCs to improve game run-time performance, and routinely employ performance intensive computer games like *Crysis* and *Fry Cry 2* (both of which can be modded) to demonstrate and document the results of PC systems before and after modding.
- 17 This kind of technological expression, identification, and eros is dramatized as popular culture in feature films like *The Fast and Furious* series (2001-2009), and earlier in *American Graffiti* (1973), *Two-Lane Blacktop* (1971), and *Hot Rod Girl* (1956).
- 18 See photos of examples from Quake2009 at <http://news.bigdownload.com/photos/quakecon-2009-pc-case-mods-pictures/2207346/> or from 2008 at <http://news.bigdownload.com/photos/quakecon-2008-pc-case-mod-gallery/955134/>, accessed 30 March 2010.
- 19 It should also be noted that in response to such dissemination of knowledge for how to hack into the Xbox system, that Microsoft invested a great deal of effort and resources to insure its subsequent Xbox 360 game platform could not be so readily hacked, and it seems to have mostly succeeded in these efforts, though to the disadvantage of its users, who must consequently look elsewhere to learn about reverse engineering and platform hacking as a basis for system innovation or customization. In this regard, Microsoft's actions may suggest it does not trust the consumers of its products to engage in activities that might improve their personal situation, but instead acts to protect what Microsoft perceives to be its corporate interests within its dispersed intellectual property.
- 20 Such practice is to be distinguished from suspect or illegal misappropriation of trade secrets through covert acquisition of materials or documents from insiders or other illicit sources, rather than as a consumer who is inquisitive and willing to void their product warranty to look inside the back box to see what's there, and who is willing to exercise the freedom to figure out how it does or doesn't operate, and who can create or modify what is found to achieve new operational capabilities.
- 21 Console hardware may also be modded through the incorporation of custom integrated circuits ("modchips") which circumvent vendor-imposed limitations or copy protection mechanisms. For example, see <http://www.modchip.com>, <http://www.mod-chip.com>, <http://www.modchipstore.com>, or <http://en.wikipedia.org/wiki/Modchip> among others, accessed 30 March 2010.
- 22 Prior to the release of WoW, Blizzard Entertainment actively pursued and ultimately shut down the independent development of a Battel.Net server portal called bnetd.org that was claimed to have been reverse engineered from public information. But Blizzard successfully prosecuted that such modding entailed work-around or breakage of in-game anticircumvention mechanism, and that such infringement made the bnetd.org modders legally culpable. The Court then ruled in favor of Blizzard Entertainment and bnetd.org was shut down. See Davidson & Assoc. et al., v. Jung et al., 422 F.3d 630, 633 (8th. Cir. 2005).
- 23 Kücklich [2005] reports the EULA for the Half-Life SDK "...hereby grants Licensee a nonexclusive, royalty-free, terminable, worldwide, non-transferable license to: (a) use, reproduce and modify the SDK in source code form, solely to develop a Mod; and (b) reproduce, distribute and license the Mod in object code form, solely to licensed end users of Half-Life, without charge."

- 24 Information on the Mozilla tri-license for the Firefox source code (the MPL as a minimally restricted license, the GPLv2 as a reciprocal license, and the LGPLv2.1 as a library-style license) can be found at <http://www.mozilla.org/MPL/>, accessed 30 March 2010.
- 25 See the Open Gaming Foundation at <http://www.opengamingfoundation.org/licenses.html> and also the d20 Game System license from the Wizards of the Coast at <http://www.wizards.com/default.asp?x=d20/welcome>, accessed 30 March 2010.
- 26 In a game like *Doom*, this internal representation is denoted by a WAD file [Kushner 2003, Morris 2003].
- 27 See <http://www.worldofwarcraft.com/ui/> and <http://www.worldofwarcraft.com/policy/ui.html>, accessed 30 March 2010.
- 28 See <http://nwn.bioware.com/builders/>, accessed 30 March 2010.
- 29 See <http://www.udk.com/>, accessed 30 March 2010.
- 30 See <http://www.garrysmud.com/>, accessed 30 March 2010.
- 31 See <http://www.idsoftware.com/business/techdownloads/>, accessed 30 March 2010.
- 32 The game development studio, Crytek, has begun to offer job interviews at their company as a prize for winning a game modding contest that they host. Further information at <http://crymod.com/thread.php?threadid=56783>, accessed 30 March 2010.
- 33 For example, the lead designer of the *Civilization V* game from Firaxis Studios, Jon Shaffer, started his career as a modder of earlier *Civilization* games. His mods were recognized as being of sufficient quality, and his academic studies in European, 19th Century German, and World War II history aligned with his *Civilization* modding interests. Subsequently, he was recruited to work at Firaxis, and has become a lead designer there. See [Murdoch and Wilson 2010] for details.
- 34 Though now moved into archived status, the description of The Layman's Guide to Mod Making at http://wiki.beyondunreal.com/Legacy:Making_Mods details what's involved in team-based mod making efforts. Similarly, Postigo's [2007] study of the Home Front Mod team for *BattleField 1942* identifies more than 25 team members occupying 18 self-selected roles in mod making.
- 35 See <http://www.planeshift.it/recruitment.html> (accessed 30 March 2010) or [Scacchi 2004] for details.
- 36 See Scacchi [2004], p.64. Also see where such encouragement in modding as a career development strategy is reiterated at <http://www.makesomethingunreal.com/overview.aspx>, where it is stated, "...Nearly half of the people who work on Epic Games' development team were former mod-makers! We want to help you make it to the big leagues as well. Think you have what it takes? Check out the [Epic Jobs](#) page." accessed 30 March 2010.
- 37 The development of the *Chex Quest* game conversion by a small game studio did allow for one of its first-time mod maker's, Chuck Jacobi, to subsequently become lead game artist for Electronic Arts LA, a major game studio. See [Lathi 2009] for details.
- 38 Scacchi [2008] also draws comparison of the world of computer games, FOSS, and scientific research computing.
- 39 For an introduction, history, and sample practices of the Warez scene, see http://en.wikipedia.org/wiki/Warez_scene, <http://www.warezscene.org/>, <http://thewarezscene.org/forums/>, or <http://torrentfreak.com/interview-with-a-warez-scene-releaser/>, accessed 30 March 2010.
- 40 See <http://torrentfreak.com/top-10-most-pirated-games-of-2008-081204/> for 2008, and for 2009 see <http://torrentfreak.com/the-most-pirated-games-of-2009-091227/>, accessed 30 March 2010.
- 41 As another major area of FOSS development activity (at SourceForge and elsewhere) is in the domain of Science/Engineering, we should not be surprised to soon see new FOSS game engines or game physics libraries being modded to support more complex physical computations beyond forces and motions, and eventually include electro-

magnetism, quantum chromodynamics, and celestial mechanics, as well as chemical, biological, geological, and meteorological processes. Conversely, as such capabilities come into being, then we expect scientists and students working in such areas to increasingly see their experimental studies employing these new science game mods [Mayo 2007].

References

Thomas Alspaugh, Hazel Asuncion, and Walt Scacchi, 2009, "Intellectual Property Rights Requirements for Heterogeneously Licensed Systems," in *Proc. 17th. Intern. Conf. Requirements Engineering (RE09)*, Atlanta, GA, ACM Press, 24-33, September 2009.

Yochai Benkler, 2006. *The Wealth of Networks: How Social Production Transforms Markets and Freedom*, Yale University Press, New Haven, CT.

Mia Consalvo, 2007. *Cheating: Gaining Advantage in Videogames*, MIT Press, Cambridge, MA.

Rob Crossley, 2008. "The Most Pirated Games in 2008," *EDGE*, 8 December 2008. <http://www.next-gen.biz/news/the-most-pirated-games-2008>, accessed 23 December 2009.

Paul Dourish, 2001. *Where the Action Is: The Foundations of Embodied Interaction*, MIT Press, Cambridge, MA.

Margaret S. Elliott and Kenneth L. Kraemer (Eds.), 2008. *Computerization Movements and Technology Diffusion: From Mainframes to Ubiquitous Computing*, ASIST Monograph Series, Information Today Inc., Medford, New Jersey.

Magy Seif El-Nasr and Brian K. Smith, "Learning Through Game Modding", *ACM Computers in Entertainment*, 4(1). Article 3B.

James Paul Gee, 2008. "Video Games and Embodiment," *Games and Culture*, Jul 2008; vol. 3: pp. 253–263.

Andy Greenburg and Mary Jane Irwin, 2008. "Spore's Piracy Problem," *Forbes.com* http://www.forbes.com/2008/09/12/spore-drm-piracy-tech-security-cx_ag_mji_0912spore.html?partner=alerts, accessed 23 December 2009.

Elizabeth Hayes and I.A. Games, 2008. "Making Computer Games and Design Thinking: A Review of Current Software and Strategies," *Games and Culture* Jul 2008; vol. 3: pp. 309 - 332.

Il-Horn Hann, Jeffrey Roberts, Sandra Slaughter, and Roy Fielding, 2002. "Economic Incentives for Participating in Open Source Software Projects," in *Proc. Twenty-Third Intern. Conf. Information Systems*, 365-372.

Andrew “bunnie” Huang, 2003. *Hacking the Xbox: An Introduction to Reverse Engineering*, No Starch Press, San Francisco, CA.

Lars Bo Jepperson, 2005, “User Toolkits for Innovation: Consumers Support Each Other,” *Journal of Product Innovation Management*. 22, 347–362.

Rob Kling and Walt Scacchi, 1982, “The Web of Computing: Computer Technology as Social Organization”, in M. Yovits, (Ed.), *Advances in Computers*, Vol 21, 3-75.

Yong Ming Kow and Bonnie Nardi, 2010. “Culture and Creativity: *World of Warcraft* Modding in China and the US,” in W.S. Bainbridge (Ed.), *Online Worlds: Convergence of the Real and the Virtual*, Springer, New York. 21-42, (to appear).

Julian Kücklich, 2005. “Precarious playbour: Modders and the digital games industry,” *Fibreculture*, number 5, at <http://journal.fibreculture.org/issue5/kucklich.html>, accessed 22 December 2009.

David Kushner, 2003. *Masters of Doom: How Two Guys Created an Empire and Transformed Pop Culture*, Random House, New York.

Evan Lathi, 2009. Chex Quest: Never Forget, *PC Gamer*, 190(16), August, p. 96.

Bruno Latour, 2005, *Reassembling the Social: An Introduction to Actor-Network Theory*, Oxford University Press, New York.

Henry Lowood, 2008. “Found Technology: Players as Innovators in the Making of Machinima.” in Tara McPherson (Ed.), *Digital Youth, Innovation, and the Unexpected*. The John D. and Catherine T. MacArthur Foundation Series on Digital Media and Learning. MIT Press, Cambridge, MA, 165–196.

Paul Marino, 2004. *3D Game-Based Filmmaking: The Art of Machinima*. Paraglyph Press, Scottsdale, AZ.

Merrilea J. Mayo, 2007. “Games for Science and Engineering Education,” *Communications of the ACM*, 50(7), 31-35, July.

Nick Montfort and Ian Bogost, 2009. *Racing the Beam: The Atari Video Computer System*, MIT Press, Cambridge, MA.

Sue Morris, 2003. “WADs, Bots, and Mods: Multiplayer FPS Games and Co-creative Media,” *Level Up Conference Proceedings: 2003 Digital Games Research Association Conference*, Utrecht, University of Utrecht, November.

Julian Murdoch and Jason Wilson, 2010. “The Education of Civilization V,” *GamePro*, 259, 59-69, April.

David Nieborg, 2005. “Am I mod or not? — An analysis of first person shooter modification culture,” paper presented at *Creative Gamers Seminar — Exploring Participatory Culture in Gaming*,

University of Tampere, Finland (14–15 January).

Joseph Pine, 1992. *Mass Customization: The New Frontier in Business Competition*. Harvard Business School Press, Boston, MA.

Alejandro Portes, 1998. "Social Capital: Its Origins and Application in Modern Sociology," *Annual Review of Sociology*, 24, 1-24.

Hector Postigo, 2007. "Of Mods and Modders: Chasing Down the Value of Fan-Based Digital Game Modifications," *Games and Culture*, Oct 2007; vol. 2: pp. 300 – 313.

Hector Postigo, 2008. Video game appropriation through modifications: Attitudes concerning intellectual property among modders and fans. *Convergence*, 14, 59–74.

Walt Scacchi, 2004. "[Free/Open Source Software Development Practices in the Computer Game Community](#)," *IEEE Software*, 21(1), 59-67, January/February 2004.

Walt Scacchi, 2007. "Free/Open Source Software Development: Recent Research Results and Emerging Opportunities," *Proc. European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, 459-468, September 2007.

Walt Scacchi, 2008. "Emerging Patterns of Intersection and Segmentation when Computerization Movements Interact," in M.S. Elliott and K.L. Kraemer (Eds.), *Computerization Movements and Technology Diffusion: From Mainframes to Ubiquitous Computing*, ASIST Monograph Series, Information Today, Inc., Medford, New Jersey, 381-404.

Walt Scacchi, 2010, "Game-Based Virtual Worlds as Decentralized Virtual Activity Systems," in W.S. Bainbridge (Ed.), *Online Worlds: Convergence of the Real and the Virtual*, Springer, New York. 223-234, (to appear).

Walt Scacchi, Robert Nideffer, and Joe Adams, 2008. "A Collaborative Science Learning Game Environment for Informal Science Education: *DinoQuest Online*," in P. Ciancarini, R. Nakatsu, M. Rauterberg, M. Roccetti (Eds.), *New Frontiers for Entertainment Computing*, IFIP International Federation for Information Processing Series, Volume 279; Boston: Springer, 71–82.

Bart Simon 2007. "Geek Chic: Machine Aesthetics, Digital Gaming, and the Cultural Politics of the Case Mod", *Games and Culture*, Jul 2007; vol. 2, 175-193.

Olli Sotamaa, 2007. "On Modder Labour, Commodification of Play, and Mod Competitions," *First Monday*, 12(9).

Lucy Suchman, 2007. *Human-Machine Reconfigurations: Plans and Situated Actions (2nd Edition)*, Cambridge University Press, New York.

Susan Leigh Star and Amselm Strauss, 1999. "Layers of Silence, Arenas of Voice: The Ecology

of Visible and Invisible Work”, *Computer-Supported Cooperative Work*, 8(1/2), 9-30, March 1999.

T. L. Taylor, 2006. “Does WoW Change Everything?: How a PvP Server, Multinational Player Base, and Surveillance Mod Scene Caused Me Pause”, *Games and Culture*, Oct 2006; vol. 1: pp. 318-337.

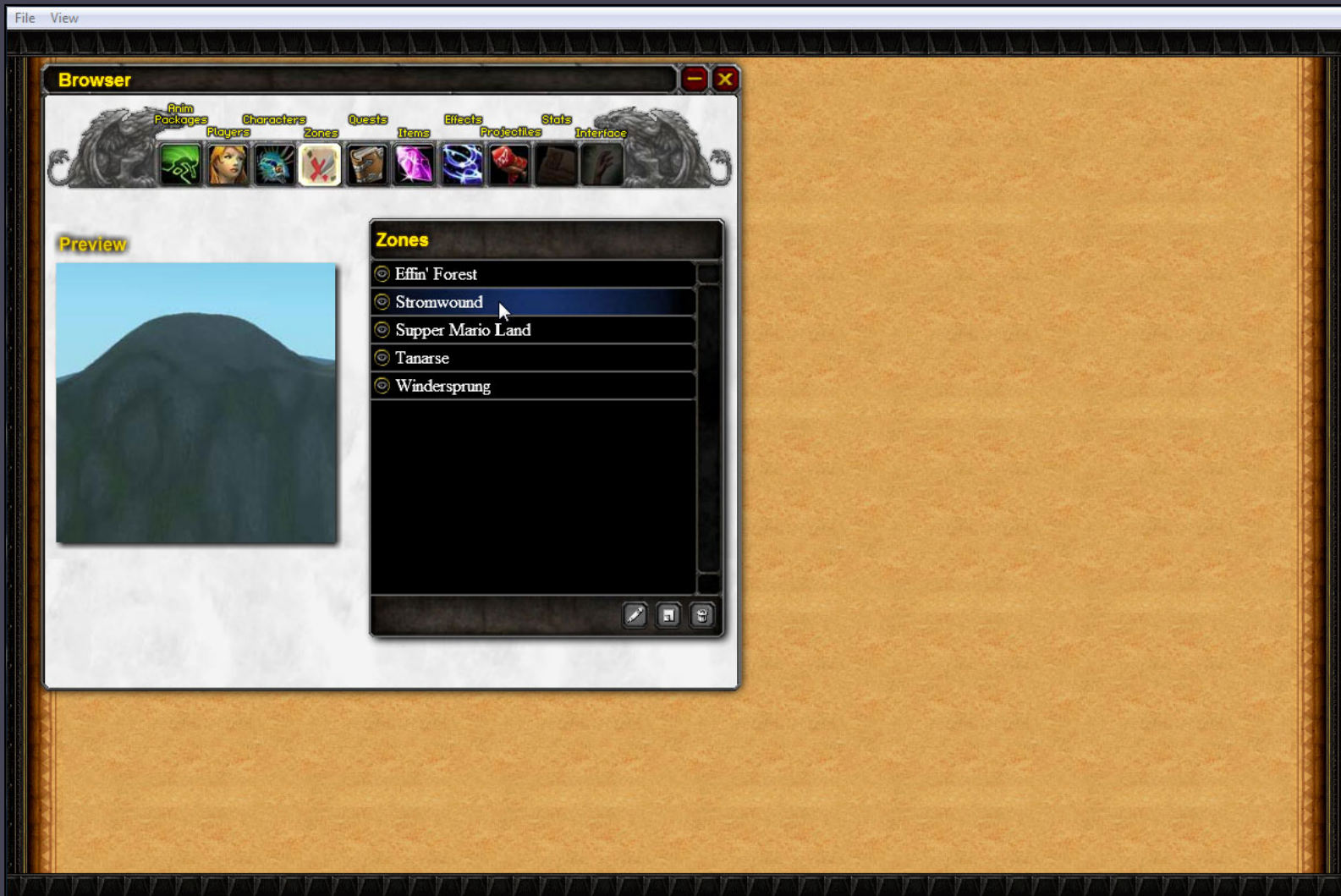
T.L. Taylor, 2009. “The Assemblage of Play,” *Games and Culture*, 4(1), 331-339, October.

Eric von Hippel and Ralph Katz, 2002. “Shifting innovation to users via toolkits,” *Management Science*, 48(7), 821-833.

"!" - Game Modding and Development Kit

(A Work Nearly Done)
'08-'10

Asset Browser



Zoom Image

- WoW inspired side-scrolling action RPG game modding and development environment
- Built in Flash using Adobe Air
- Opens to a base project canvas that manages multiple game editing windows
- Asset Browser previews all game assets and launches editors

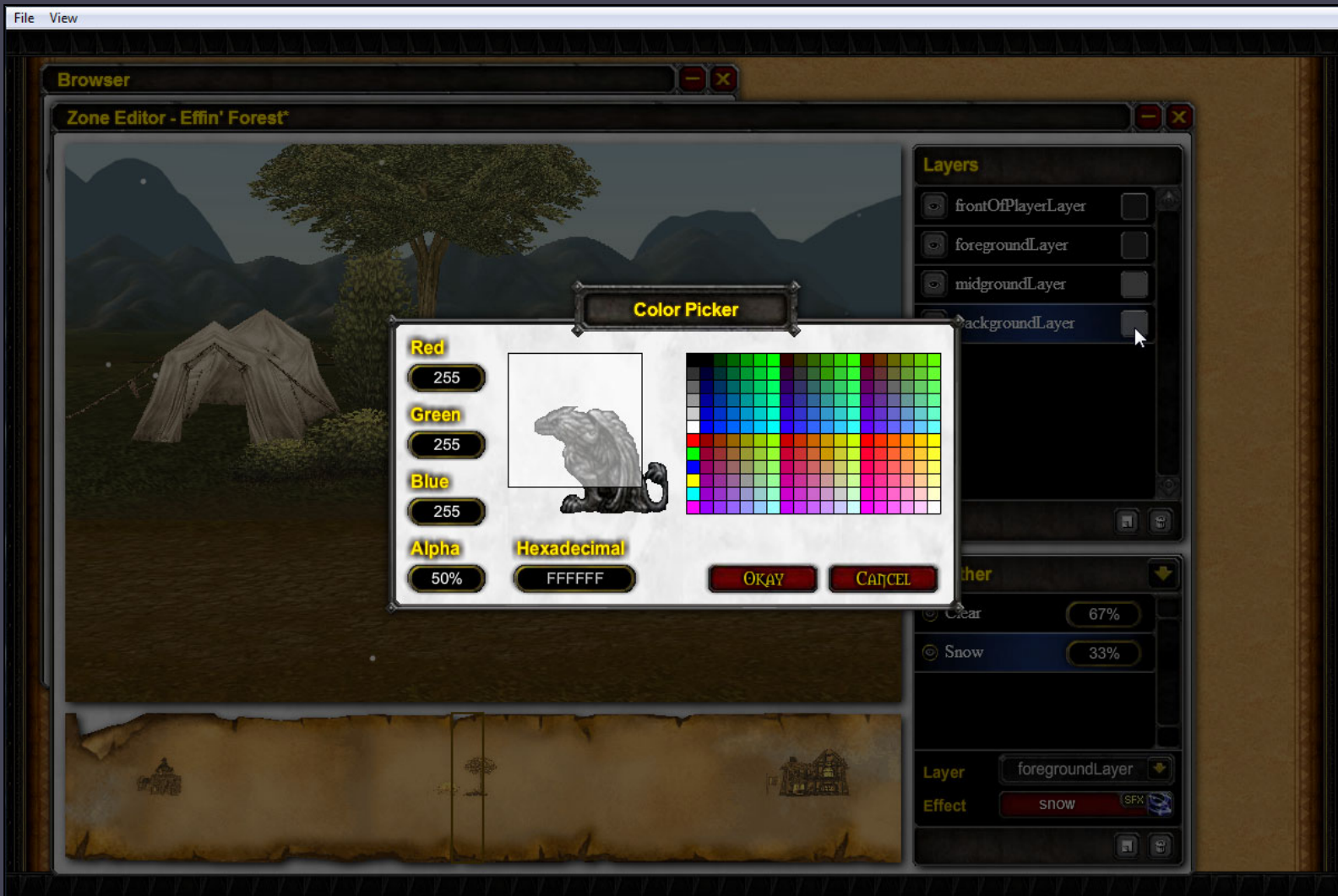
Level Editor



Zoom Image

- Photoshop style editing mode
- Import any number of layers to place objects
- Adjust scroll speed of layers independently for parallax scrolling
- Create custom weather conditions with a percent chance of occurring
- Click and drag to scroll zone and place objects, characters, spawn points, spawn conditions, etc
- Minimap supports placement and manipulation of objects as well

Color Picker



Zoom Image

- Assign each layer unique graphics and color tinting
- Assign key-framed time of day tinting changes based on a 24 hour clock that automatically interpolates color values

Quest Editor



Zoom Image

- Supports creation of multiple quest types - collect, kill, escort, object initiated, chained, etc
- Establish requirements for access, rewards, items needed for completion, and more
- Define relationships to other quests
- Create quest text for introduction, objectives, progress, completion

Item Editor



Zoom Image

- Create items associated with quests and random drops
- Set descriptive "flavor text"
- Assign quality, type, subtype, level requirements, cost, bonuses, ability cost/effects

Ability Setting



Zoom Image

- Set item abilities such as type, whether item effects stack, stack limits, duration
- Link associated graphics
- Create descriptive text summarizing abilities for player

Requirements Setting



Zoom Image

- Establish hierarchies of requirements that must be met prior to items and quests becoming available

Character Editor



Zoom Image

- Create custom characters
- Choose attributes and behaviors based on pre-defined templates or create your own
- Establish identity markers (faction, profession, race, class, and anything else you want)
- Attach pre-set animations or create your own
- Create character tinting effects based on state
- Determine default equipped gear
- Author chat dialogs
- Define complex sets of behaviors
- Set loot tables
- Decide if character is a quest giver, and which quests are available if so
- Preview the settings

Projectile Editor



Zoom Image

- Choose from a seed particle list to build more complex projectile effects
- Set properties for projectiles (name, duration, damage, trail effects, movement patterns, etc)
- Preview the settings

Particle Effects Editor



Zoom Image

- Choose from a seed particle list to build more complex particle effects
- Associate these effects to projectiles or other items and conditions
- Set properties for particles (whether static or dynamic, behavior, appearance, amount, duration, etc)
- Link custom made sounds to particle effects
- Preview the settings

Animation Editor



Zoom Image

- Browse and link a series of image frames to custom character animations
- Associate animations with behavior states (idle, jump, attack, stun, death, cast, etc)
- Define which frames get used
- Set bounding boxes to establish impact areas
- Preview the settings

Character Action Editor



Zoom Image

- Define specific actions to use to control custom NPC behaviors
- Tie actions to custom animations and specify primary and secondary use types
- Adjust interaction settings and effects
- Specify movement speed and angle, whether action is interruptible, sounds, and particle effects
- Set impact effects including damage multipliers, range, force, angle, and special effects

Character Behavior Editor - Flow View



Zoom Image

- Create complex NPC behaviors and scripted events
- View behaviors in "Flow" mode (diagram) or "Full" mode (code)
- Define custom states for NPCs to switch to based on action conditions

Character Behavior Editor - Full View



Zoom Image

- Set NPC faction based on specific states
- Set NPC auras (ground color upon selection) independant of faction
- Link a variety of predefined conditions to code blocks
- Easy enough for non-programmers to use

Character Dialog Editor



Zoom Image

- Define complex NPC and player dialog trees
- Trigger dialogs based on custom state lists
- Allows NPCs to "make" player character speak/respond
- Establish percent likelihood of dialog to occur

Character Quest Handling Editor



Zoom Image

- Determine which quests get handled by particular NPCs
- Define whether NPC gives and /or finishes a specific quest, and /or serves as a progress checker
- Create custom greetings for quest-handling NPCs

Character Equipment Editor



Zoom Image

- Link items to characters
- Create custom abilities for characters (similar to the WoW spellbook)
- View the combined stats of characters as equipped

Player Editor



Zoom Image

- Define characters to use as main player character (up to 4)
- Create custom identity markers that can be tag-referenced to display in quest text to create a personalized feel
- Define a special login animation for playable characters to display on startup
- Link unique background graphics and icons to playable character login screen
- Select unique starting for each playable character
- Set custom player capabilities such as run, jump, and attack and dodge velocity / angle

PORTAL

Understanding the Role of Licenses and Evolution in Open Architecture Software Ecosystems

Walt Scacchi

Institute for Software Research. University of California, Irvine USA

Thomas A. Alspaugh

Computer Science Dept., Georgetown University, Washington, DC, USA

Abstract

The role of software ecosystems in the development and evolution of heterogeneously-licensed open architecture systems has received insufficient consideration. Such systems are composed of components potentially under two or more licenses, open source or proprietary or both, in an architecture in which evolution can occur by evolving existing components, replacing them, or refactoring. The software licenses of the components both facilitate and constrain the system's ecosystem and its evolution, and the licenses' rights and obligations are crucial in producing an acceptable system. Consequently, software component licenses and the architectural composition of a system determine the software ecosystem niche where a system lies. Understanding and describing software ecosystem niches is a key contribution of this work. A case study of an open architecture software system that articulates different niches is employed to this end. We examine how the architecture and software component licenses of a composed system at design-time, build-time, and run-time help determine the system's software ecosystem niche and provide insight and guidance for identifying and selecting potential evolutionary paths of system, architecture, and niches.

Keywords: Software architecture, software ecosystems, software licenses, open

Email addresses: wscacchi@ics.uci.edu (Walt Scacchi), alspaugh@cs.georgetown.edu (Thomas A. Alspaugh)

1. Introduction

A substantial number of development organizations are adopting a strategy in which a software-intensive system (one in which software plays a crucial role) is developed with an *open architecture* (OA) [29], whose components may be open source software (OSS) or proprietary with open application programming interfaces (APIs). Such systems evolve not only through the evolution of their individual components, but also through replacement of one component by another, possibly from a different producer or under a different license. With this approach, another organization often comes between software component producers and system consumers. These organizations take on the role of system architect or integrator, either as independent software vendors, government contractors, system integration consultants, or in-house system integrators. In turn, such an integrator designs a system architecture that can be composed of components largely produced elsewhere, interconnected through interfaces accommodating use of dynamic links, intra- or inter-application scripts, communication protocols, software buses, databases/repositories, plug-ins, libraries or software shims as necessary to achieve the desired result. An OA development process results in an ecosystem in which the integrator is influenced from one direction by the goals, interfaces, license choices, and release cycles of the software component producers, and from another direction by the needs of the system's consumers. As a result the software components are reused more widely, and the resulting OA systems can achieve reuse benefits such as reduced costs, increased reliability, and potentially increased agility in evolving to meet changing needs. An emerging challenge is to realize the benefits of this approach when the individual components are *heterogeneously licensed* [2, 17, 33], each potentially with a different license, rather than a single OSS license as in uniformly-licensed OSS projects or a single proprietary license as in proprietary development.

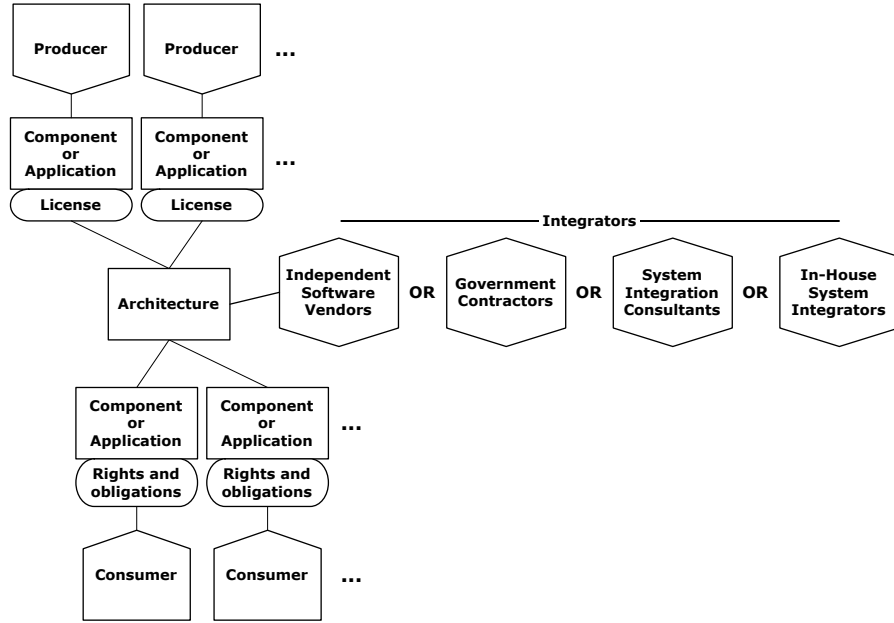


Figure 1: Schema for software supply networks

This challenge is inevitably entwined with the software ecosystems that arise for OA systems (Figure 1). We find that an OA software ecosystem involves organizations and individuals producing, composing, and consuming components that articulate software supply networks from producers to consumers, but also

- the OA of the system(s) in question,
- the open interfaces met by the components,
- the degree of coupling in the evolution of related components, and
- the rights and obligations resulting from the software licenses under which various components are released, that propagate from producers to consumers.

A motivating example of this approach is the Unity game development tool, produced by Unity Technologies [35]. Its license agreement, from which we quote below, lists eleven distinct licenses and indicates the tool is produced, apparently

using an OA approach, using at least 18 externally produced components or groups of components:

1. The Mono Class Library, Copyright 2005-2008 Novell, Inc.
2. The Mono Runtime Libraries, Copyright 2005-2008 Novell, Inc.
3. Boo, Copyright 2003-2008 Rodrigo B. Oliveira
4. UnityScript, Copyright 2005-2008 Rodrigo B. Oliveira
5. OpenAL cross platform audio library, Copyright 1999-2006 by authors.
6. PhysX physics library. Copyright 2003-2008 by Ageia Technologies, Inc.
7. libvorbis. Copyright (c) 2002-2007 Xiph.org Foundation
8. libtheora. Copyright (c) 2002-2007 Xiph.org Foundation
9. zlib general purpose compression library. Copyright (c) 1995-2005 Jean-loup Gailly and Mark Adler
10. libpng PNG reference library
11. jpeglib JPEG library. Copyright (C) 1991-1998, Thomas G. Lane.
12. Twilight Prophecy SDK, a multi-platform development system for virtual reality and multimedia. Copyright 1997-2003 Twilight 3D Finland Oy Ltd
13. dynamic_bitset, Copyright Chuck Allison and Jeremy Siek 2001-2002.
14. The Mono C# Compiler and Tools, Copyright 2005-2008 Novell, Inc.
15. libcurl. Copyright (c) 1996-2008, Daniel Stenberg <daniel@haxx.se>.
16. PostgreSQL Database Management System
17. FreeType. Copyright (c) 2007 The FreeType Project (www.freetype.org).
18. NVIDIA Cg. Copyright (c) 2002-2008 NVIDIA Corp.

The software ecosystem for Unity3D as a standalone software package is delimited by the diverse set of software components listed above. However the architecture that integrates or composes these components is closed and thus unknown to a system consumer, as is the manner in which the different licenses associated with these components impose obligations or provide rights to consumers, or on the other components to which they are interconnected. Subsequently, we see that software ecosystems can be understood in part by examining relationships between architectural composition of software components that are subject to different licenses, and this necessitates access to the system's architecture composition. By examining the open architecture of a specific composed software system, it becomes possible to explicitly identify the software ecosystem niche in which the system is embedded.

A software ecosystem constitutes a software supply network that connects software producers to integrators to consumers through licensed components

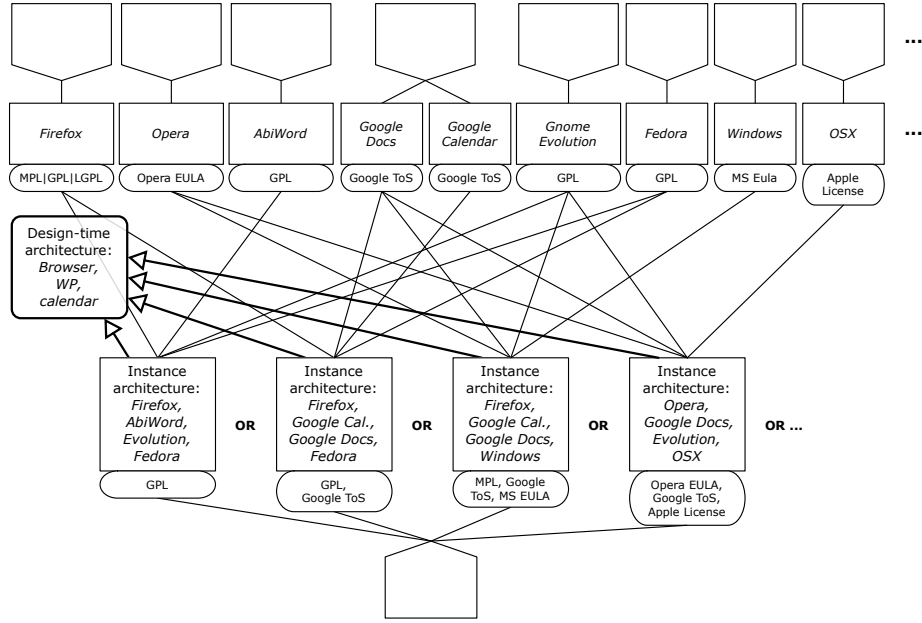


Figure 2: Ecosystem for three possible instantiations of a single design architecture

and composed systems. Figure 2 outlines the software ecosystem elements and relationships for an OA case study that we examine in this paper.

A *software ecosystem niche* articulates a specific software supply network that interconnects particular software producers, integrators, and consumers. A software system defined niche may lie within an existing single ecosystem, or one that may span a network of software producer ecosystems. A composed software system architecture helps determine the software ecosystem niche, since the architecture identifies the components, their licenses and producers, and thus the network of software ecosystems in which it participates. Such a niche also transmits license-borne obligations and access/usage rights passed from the participating software component producers through integrators and to system consumers. Thus, system architects or component integrators help determine in which software ecosystem niche a given instance architecture for the system participates.

As a software system evolves over time, as its components are updated or

changed, or their architectural interconnections are refactored, it is desirable to determine whether and how the system’s ecosystem niche may have changed. A system may evolve because its consumers want to migrate to alternatives from different component producers, or choose components whose licenses are more/less desirable. Software system consumers may want to direct their system integrators to compose the system’s architecture so as to move into or away from certain niches. Consequently, system integrators can update or modify system architectural choices at design-time, build-time (when components are compiled together into an executable), or run-time (when bindings to remote executable services are instantiated) to move a software system from one niche to another. Thus, understanding how software ecosystem niches emerge is a useful concept that links software engineering concerns for software architecture, system integration/composition, and software evolution to organizational and supply network relationships between software component producers, integrators and system consumers.

To help explain how OA systems articulate software ecosystem niches, we provide a software architecture case study for use in this paper. This system is intentionally simple for expository purposes. Its architectural design composes a web browser, word processor, calendaring and email applications, host platform operating system, and remote services as its components. Equivalent components from different OSS or proprietary software producers can be identified, where each alternative is subject to a different type of software license. For example, for Web browsers, we consider the Firefox browser from the Mozilla Foundation, which comes with a choice of OSS license (MPL, GPL, or LGPL), and the Opera browser from Opera Software, which comes with a proprietary software end-user license agreement (EULA). Similarly, for word processor, we consider the OSS AbiWord application (GPL) and Web-based Google Docs service (proprietary Terms of Service). The OA we describe covers a number of systems we have identified, built, and deployed in a university research laboratory. Example niches involving these components appear in Figures 3 and 4. We have also developed OA systems with more complex architectures that in-

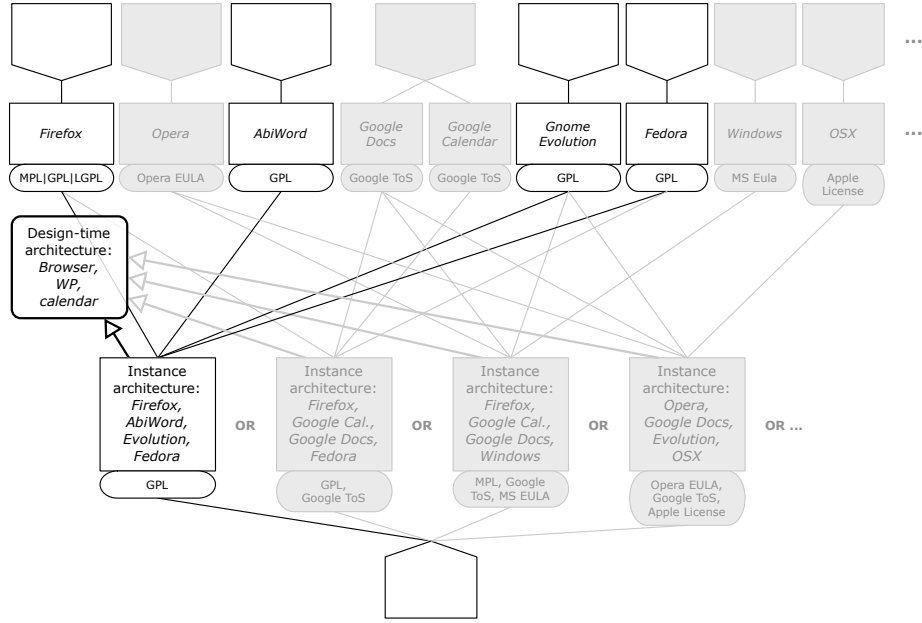


Figure 3: The ecosystem niche for one instance architecture

corporate components for content management systems (Drupal), wikis (MediaWiki), blogs (B2evolution), teleconferencing and media servers (Flash media server, Red5 media server), chat (BlaB! Lite), Web spiders and search engines (Nutch, Lucene, Sphider), relational database management systems (MySQL), and others. Furthermore, the OSS application stacks and infrastructure (platform) stacks found at BitNami.org/stacks (accessed 29 April 2010) could also be incorporated in OA systems, as could their proprietary counterparts. However, these more complex OAs still reflect the core architectural concepts and constructs, as well as software ecosystem relationships, that we present in our example case study in a simpler manner.

The software ecosystem niches for the case study system, or indeed any system, depend on which component implementations are used and the architecture in which they are combined and instantiated, as does the overall rights and obligations for the instantiated system. In addition, we build on previous work on heterogeneously-licensed systems [2, 17, 33] by examining how OA development

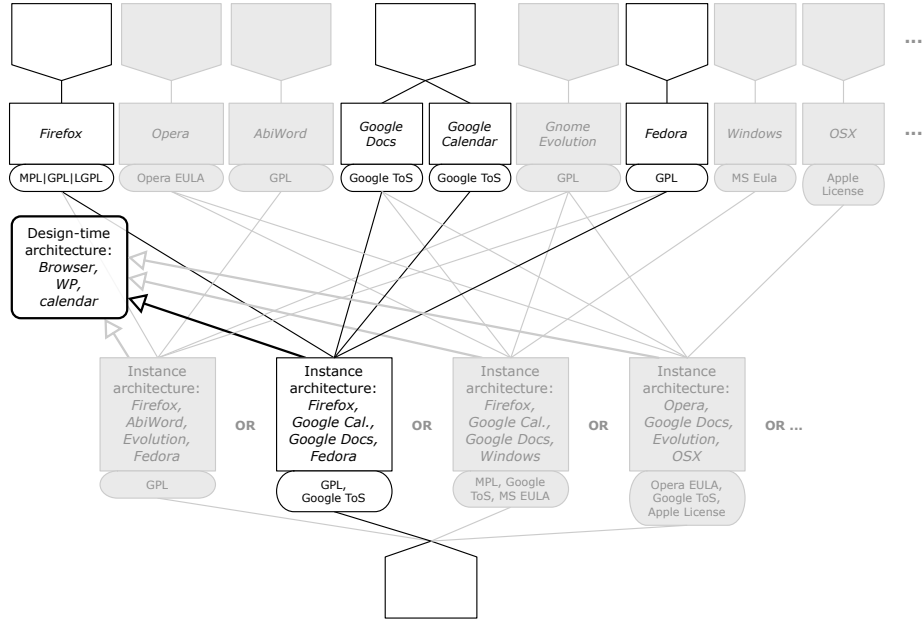


Figure 4: The ecosystem niche for a second instance architecture

affects and is affected by software ecosystems, and the role of component licenses in shaping OA software ecosystem niches.

Consequently, we focus our attention to understand the ecosystem of an open architecture software system such that:

- It must rest on a license structure of rights and obligations (Section 4), focusing on obligations that are enactable and testable¹.
- It must take account of the distinctions between the design-time, build-time, and distribution-time architectures (Section 3) and the rights and obligations that come into play for each of them.

¹For example, many OSS licenses include an obligation to make a component's modified code public, and whether a specific version of the code is public at a specified Web address is both enactable (it can be put into practice) and testable. In contrast, the General Public License (GPL) v.3 provision "No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty" is not enactable in any obvious way, nor is it testable — how can one verify what others deem?

- It must distinguish the architectural constructs significant for software licenses, and embody their effects on rights and obligations (Section 3).
- It must define license architectures.
- It must account for alternative ways in which software systems, components, and licenses can evolve (Section 5), and
- It must provide an automated environment for creating and managing license architectures. We are developing a prototype that manages a license architecture as a view of the system architecture [3].

The remainder of this paper is organized as follows. Section 2 places this work in the context of related research. Section 3 discusses open architecture, and the influence of software licenses is discussed in Section 4. Section 5 addresses evolution of software ecosystems. Section 6 discusses some implications that follow from this study, and Section 7 concludes the paper.

2. Related Research

The study of software ecosystems is emerging as an exciting new area of systematic investigation and conceptual development within software engineering. Understanding the many possible roles that software ecosystems can play in shaping software engineering practice is gaining more attention since the concept first appeared [23]. Bosch [7] builds a conceptual lineage from software product line (SPL) concepts and practices [6, 13] to software ecosystems. SPLs focus on the centralized development of families of related systems from reusable components hosted on a common platform with an intra-organizational base, with the resulting systems either intended for in-house use or commercial deployments. Software ecosystems then are seen to extend this practice to systems hosted on an inter-organizational base, which may resemble development approaches conceived for virtual enterprises for software development [26]. Producers of commercial software applications or packages thus need to adapt their development strategy and business model to one focused on coordinating and

guiding decentralized software development of its products and enhancements (e.g., plug-in components).

Along with other colleagues [8, 12, 36], Bosch identifies alternative ways to connect reusable software components through integration and tight coupling found in SPLs, or via loose coupling using glue code, scripting or other late binding composition schemes in ecosystems or other decentralized enterprises [26, 27], as a key facet that can enable software producers to build systems from diverse sources.

Jansen and colleagues [18, 19] draw attention to their observation that software ecosystems (a) embed software supply networks that span multiple organizations, and (b) are embedded within a network of intersecting or overlapping software ecosystems that span the world of software engineering practice. Scacchi [32] for example, identifies that the world of open source software (OSS) development is a software ecosystem different from those of commercial software producers, and its supply networks are articulated within a network of FOSS development projects. Networks of OSS ecosystems have also begun to appear around very large OSS projects for Web browsers, Web servers, word processors, and others, as well as related application development environments like NetBeans and Eclipse, and these networks have become part of global information infrastructures [20].

OSS ecosystems also exhibit strong relationships between the ongoing evolution of OSS systems and their developer/user communities, such that the success of one co-depends on the success of the other [32]. Ven and Mannaert discuss the challenges independent software vendors face in combining OSS and proprietary components, with emphasis on how OSS components evolve and are maintained in this context [37].

Boucharas and colleagues [9] then draw attention to the need to more systematically and formally model the contours of software supply networks, ecosystems, and networks of ecosystems. Such a formal modeling base may then help in systematically reasoning about what kinds of relationships or strategies may arise within a software ecosystem. For example, Kuehnel [21] exam-

ines how Microsoft’s software ecosystem developed around in operating systems (MS Windows) and key applications (e.g., MS Office) may be transforming from “predator” to “prey” in its effort to control the expansion of its markets to accommodate OSS (as the extant prey) that eschew closed source software with proprietary software licenses.

Our work in this area builds on these efforts in the following ways. First, we share the view of a need for examining software ecosystems, but we start from software system architectures that can be formally modeled and analyzed with automated tool support [6, 34]. Explicit modeling of software architectures enables the ability to view and analyze them at design-time, build-time, or deployment/run-time. Software architectures also serve as a mechanism for coordinating decentralized software development across multi-site projects [30]. Similarly, explicit models allow for the specification of system architectures using either proprietary software components with open APIs, OSS components, or combinations thereof, thereby realizing open architecture (OA) systems [33]. We then find value in attributing open architecture components with their (intellectual property) licenses [3], since software licenses are an expression of contractual/social obligations that software consumers must fulfill in order to realize the rights to use the software in specified allowable manners, as determined by the software’s producers.

3. Open Architectures

Open architecture (OA) software is a customization technique introduced by Oreizy [29] that enables third parties to modify a software system through its exposed architecture, evolving the system by replacing its components. Increasingly more software-intensive systems are developed using an OA strategy, not only with OSS components but also proprietary components with open APIs (e.g. [35]). Using this approach can lower development costs and increase reliability and function [33]. Composing a system with heterogeneously-licensed components, however, increases the likelihood of conflicts, liabilities, and no-rights stemming from incompatible licenses. Thus, in our work we define an

OA system as a *software system consisting of components that are either open source or proprietary with open API, whose overall system rights at a minimum allow its use and redistribution, in full or in part.*

It may appear that using a system architecture that incorporate OSS components and uses open APIs will result in an OA system. But not all such architectures will produce an OA, since the (possibly empty) set of available license rights for an OA system depends on: (a) how and why OSS and open APIs are located within the system architecture, (b) how OSS and open APIs are implemented, embedded, or interconnected, and (c) the degree to which the licenses of different OSS components encumber all or part of a software system’s architecture into which they are integrated [1, 33].

The following kinds of software elements appearing in common software architectures can affect whether the resulting systems are open or closed [5].

Software source code components—These can be either (a) standalone programs, (b) libraries, frameworks, or middleware, (c) inter-application script code such as C shell scripts, or (d) intra-application script code, as for creating Rich Internet Applications using domain-specific languages such as XUL for the Firefox Web browser [15] or “mashups” [25]. Their source code is available and they can be rebuilt. Each may have its own distinct license, though often script code that merely connects programs and data flows does not, unless the code is substantial, reusable, or proprietary.

Executable components—These components are in binary form, and the source code may not be open for access, review, modification, or possible redistribution [31]. If proprietary, they often cannot be redistributed, and so such components will be present in the design- and run-time architectures but not in the distribution-time architecture.

Software services—An appropriate software service can replace a source code or executable component.

Application programming interfaces/APIs—Availability of externally visible and accessible APIs is the minimum requirement for an “open system” [24]. Open APIs are not and cannot be licensed, but they can limit the propa-

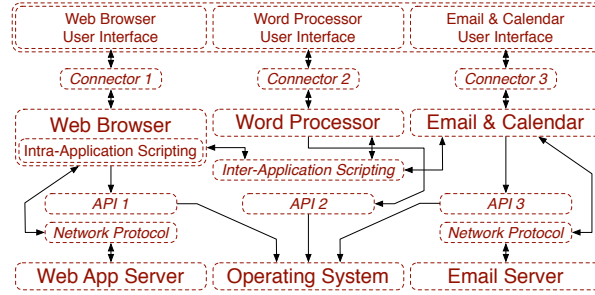


Figure 5: A design-time architecture

gation of license obligations.

Software connectors—Software whose intended purpose is to provide a standard or reusable way of communication through common interfaces, e.g. High Level Architecture [22], CORBA, MS .NET, Enterprise Java Beans, and GNU Lesser General Public License (LGPL) libraries. Connectors can also limit the propagation of license obligations.

Methods of composition—These include linking as part of a configured subsystem, dynamic linking, and client-server connections. Methods of composition affect license obligation propagation, with different methods affecting different licenses.

Configured system or subsystem architectures—These are software systems that are used as atomic components of a larger system, and whose internal architecture may comprise components with different licenses, affecting the overall system license. To minimize license interaction, a configured system or sub-architecture may be surrounded by what we term a *license firewall*, namely a layer of dynamic links, client-server connections, license shims, or other connectors that block the propagation of reciprocal obligations.

With these architectural elements, we can create an design-time or reference architecture for a system that conforms to the software supply network shown in Figure 2. This design-time architecture appears in Figure 5; note that it only specifies components by type rather than by producer, meaning the choice of producer component remains unbound at this point.

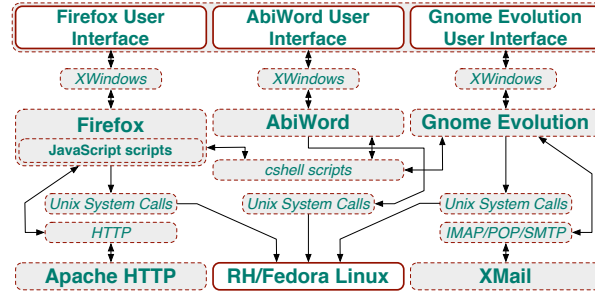


Figure 6: A build-time architecture

Then in Figure 6, we create a build-time rendering of this architectural design by selecting specific components from designated software producers. The gray boxes correspond to components and connectors not visible in the run-time instantiation of the system in Figure 7.

Figures 7, 8, and 9 display alternative run-time instantiations of the design-time architecture of Figure 5. The architectural run-time instance in Figure 7 corresponds to the software ecosystem niche shown in Figure 3; Figure 8 corresponds to the niche in Figure 4; and Figure 9 designates yet another niche different from the previous two.

Each component selection implies acceptance of the license obligations and rights that the producer seeks to transmit to the components consumers. However in an OA design development, component interconnections may be used to intentionally or unintentionally propagate these obligations onto other components whose licenses may conflict with them or fail to match [3, 17]; the system integrator can decide to insert software shims using scripts, dynamic links to remote services, data communication protocols, or libraries to mitigate or firewall the extent to which a component’s license obligations propagate. This style of build-time composition can be used to accommodate a system’s consumers’ policy for choosing systems that avoid certain licenses, or that isolate the license obligations of certain desirable components. It also allows system integrators and consumers to follow a “best of breed” policy in the selection of system components. Finally, if no license conflicts exist in the system, or if the integra-

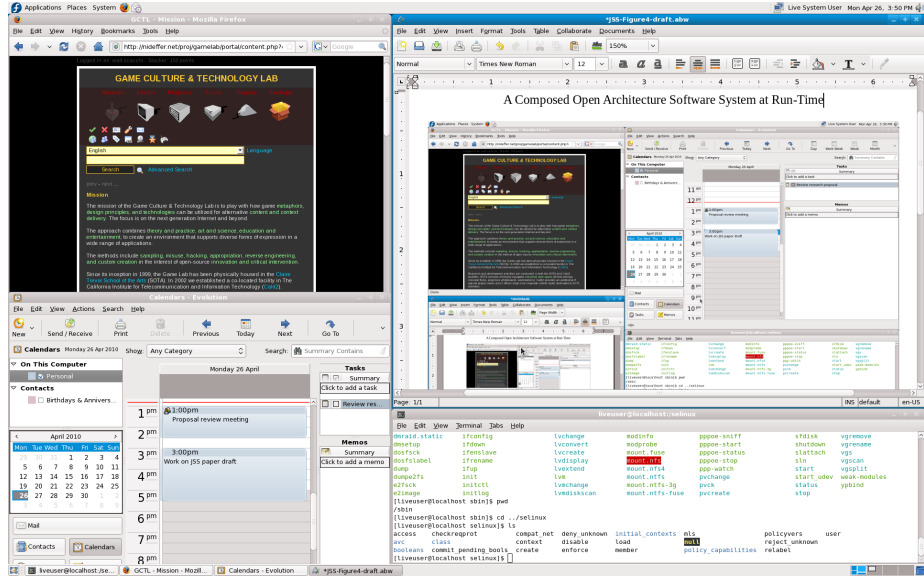


Figure 7: An instantiation at run-time (Firefox, AbiWord, Gnome Evolution, Fedora) of the build-time architecture of Figure 6 that determines the ecosystem niche of Figure 3

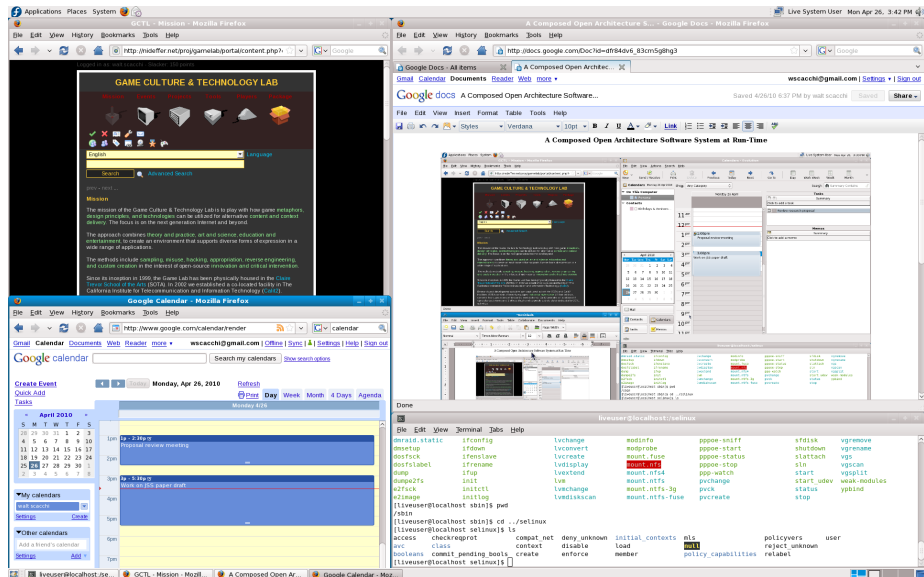


Figure 8: A second instantiation at run-time (Firefox, Google Docs and Calendar, Fedora) determining the ecosystem niche of Figure 4

and license notices.

Reciprocal OSS licenses take a more active stance towards sharing and reusing software by imposing the obligation that reciprocally-licensed software not be combined (for various definitions of “combined”) with any software that is not in turn also released under the reciprocal license. Those for which most or all ways of combining software propagate reciprocal obligations are termed *strongly reciprocal*. Examples are the GPL and the Affero GPL (AGPL). The purpose of the AGPL is to prevent a GPL software component from being integrated into an OA system as a remote server, or from being wrapped with shims to inhibit its ability to propagate the GPL obligations and rights. The purpose of these licenses is to ensure that software so licensed will maintain (and can propagate) the freedom to access, study, modify, and redistribute the software source code, which academic licenses do not. This in turn assures the access, use, and reusability of the source code for other software producers and system integrators. Those licenses for which only certain ways of combining software propagate reciprocal obligations are termed *weakly reciprocal*. Examples are the Lesser GPL (LGPL), Mozilla Public License (MPL), and Common Public License. The goals of reciprocal licensing are to increase the domain of OSS by encouraging developers to bring more components under its aegis, and to prevent improvements to OSS components from vanishing behind proprietary licenses.

Both proprietary and OSS licenses typically disclaim liability, assert no warranty is implied, and obligate licensees to not use the licensor’s name or trademarks. Newer licenses often cover patent issues as well, either giving a restricted patent license or explicitly excluding patent rights. The Mozilla Disjunctive Tri-License licenses the core Mozilla components under any one of three licenses (MPL, GPL, or the GNU Lesser General Public License LGPL); OSS developers or OA system integrators can choose the one that best suits their needs for a particular project and component.

The Open Source Initiative (OSI) maintains a widely-respected definition of “open source” and gives its approval to licenses that meet it [28]. OSI maintains

and publishes a repository of approximately 70 approved OSS licenses which tend to vary in the terms and conditions of their declared obligations and rights. However, all these licenses tend to cluster into either a strongly reciprocal, weakly reciprocal, or minimally restrictive/academic license type.

Common practice has been for an OSS project to choose a single license under which all its products are released, and to require developers to contribute their work only under conditions compatible with that license. For example, the Apache Contributor License Agreement grants enough of each author’s rights to the Apache Software Foundation for the foundation to license the resulting systems under the Apache Software License. This sort of rights regime, in which the rights to a system’s components are homogeneously granted and the system has a single well-defined OSS license, was the norm in the early days of OSS and continues to be practiced.

We have developed an approach for expressing software licenses that is more formal and less ambiguous than natural language, and that allows us to calculate and identify conflicts arising from the rights and obligations of two or more component’s licenses. Our approach is based on Hohfeld’s classic group of eight fundamental jural relations [Hohfeld 1913], of which we use right, duty, no-right, and privilege (Figure 10). We start with a tuple `<actor, operation, action, object>` for expressing a right or obligation. The actor is the “licensee” for all the licenses we have examined. The operation is one of the following: “may”, “must”, “must not”, or “need not”, with “may” and “need not” expressing rights and “must” and “must not” expressing obligations. Because copyright rights are only available to entities who have been granted a sublicense, only the listed rights are available, and the absence of a right means that it is not available. The action is a verb or verb phrase describing what may, must, must not, or need not be done, with the object completing the description. A license may be expressed as a set of rights, with each right associated with zero or more obligations that must be fulfilled in order to enjoy that right. Figure 11 sketches two rights from GPL 2.0, the first one with no obligations and the second with three corresponding obligations. Elsewhere, the details of this license specification

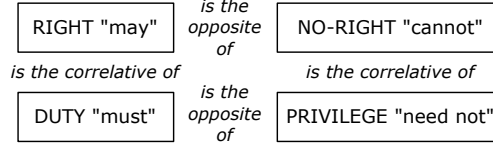


Figure 10: Hohfeld's four basic relations

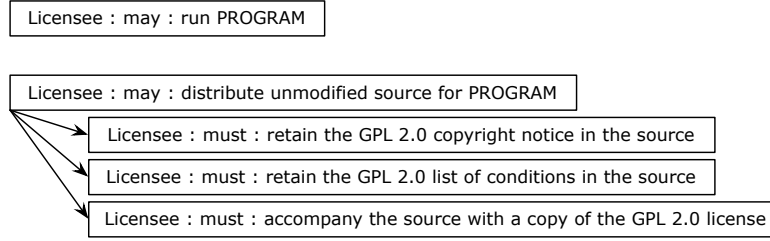


Figure 11: Tuples for some rights and obligations of the GPL 2.0 license

scheme, how it can be used to attribute software architectures to produce a system license architecture, and how it can be formalized into a semantic meta-model [3].

HLS designers have developed a number heuristics to guide architectural design while avoiding some license conflicts. First, it is possible to use a reciprocally-licensed component through a license firewall that limits the scope of reciprocal obligations. Rather than connecting conflicting components directly through static or other build-time links, the connection is made through a dynamic link, client-server protocol, license shim (such as an LGPL connector), or run-time plug-ins. A second approach used by a number of large organizations is simply to avoid using any components with reciprocal licenses. A third approach is to meet the license obligations (if that is possible) by for example retaining copyright and license notices in the source and publishing the source code. However, even using design heuristics such as these (and there are many), keeping track of license rights and obligations across components that are interconnected in complex OAs quickly becomes too cumbersome. Automated support is needed to manage the multi-component, multi-license complexity. Accordingly, we are developing an automated support capability as part of the

ArchStudio architecture design environment [14] that can analyze the addition of software license properties such as those shown in Figure 11 to the interfaces of software components in an OA system. For example, in Figure 12 we see a rendering of the OA system from our case study with the AbiWord word processing component highlighted. This component’s APIs would be attributed with the GPL license obligations and rights in Figure 11, since AbiWord is licensed under GPL, as are other components like the Gnome Evolution calendaring and email application and also the Red Hat/Fedora Linux operating system platform. As the architectural interconnections shown in the model of Figure 12 indicate that none of these components covered by GPL are directly interconnected to another licensed component, their license obligations do not propagate or become “viral” in this architectural composition. Replacing any of these GPL components with non-GPL but still OSS components would not change the total set of obligations and rights on the system with this architecture; the system would remain OSS, but the software ecosystem niche in which it resides would shift to another niche. Once again, software licenses interact with software architectures and together they help determine which software ecosystem niche will embed an instantiated run-time version of the system.

5. Architecture, License, and Ecosystem Evolution

An OA system can evolve by a number of distinct mechanisms, some of which are common to all systems but others of which are a result of heterogeneous component licenses in a single system.

By component evolution— One or more components can evolve, altering the overall system’s characteristics (for example, upgrading and replacing the Firefox Web browser from version 3.5 to 3.6). Such minor versions changes generally have no effect on system architecture.

By component replacement— One or more components may be replaced by others with modestly different functionality but similar interface, or with a different interface and the addition of shim code to make it match (for example, replacing the AbiWord word processor with either Open Office Writer or MS

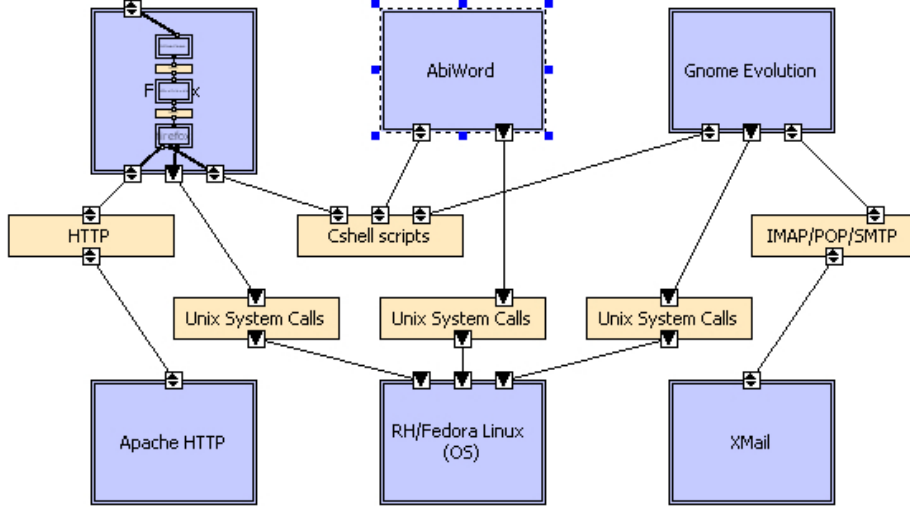


Figure 12: The model of the system architecture used in our case study as rendered in our automated tool, described elsewhere [2, 3]

Word). However, changes in the format or structure of component APIs may necessitate build-time and run-time updates to component connectors. Figure 13 shows some possible alternative system compositions that result from replacing components by others of the same type but with a different license.

By architecture evolution— The OA can evolve, using the same components but in a different configuration, altering the system characteristics. For example, as discussed in Section 4, revising or refactoring the configuration in which a component is connected can change how its license affects the rights and obligations for the overall system. This could arise when replacing word processing, calendaring, and email components and their connectors with Web-browser-based services such as Google Docs, Google Calendar, and Google Mail. The replacement would eliminate the legacy components and relocate the desired application functionality to operate within the Web browser component, resulting in what might be considered a simpler and easier-to-maintain system architecture, but one that is less open and now subject to a proprietary Terms of Service license. System consumer policy preferences for licenses, and subsequent participation in a different ecosystem niche, may thus mediate whether

such an alternative system architecture is desirable or not.

By component license evolution— The license under which a component is available may change, as for example when the license for the Mozilla core components was changed from the Mozilla Public License (MPL) to the current Mozilla Disjunctive Tri-License; or the component may be made available under a new version of the same license, as for example when the GNU General Public License (GPL) version 3 was released. The three architectures in Figure 13 that incorporate the Firefox Web browser show how its tri-license creates new evolutionary paths by offering different licensing options. These options and paths were not available previously with earlier versions of this component offered under only one or two license alternatives.

By a change to the desired rights or acceptable obligations— The OA system’s integrator or consumers may desire additional license rights (for example the right to sublicense in addition to the right to distribute), or no longer desire specific rights; or the set of license obligations they find acceptable may change. In either case the OA system evolves, whether by changing components, evolving the architecture, or other means, to provide the desired rights within the scope of the acceptable obligations. For example, they may no longer be willing or able to provide the source code for components within the reciprocity scope of a GPL-licensed module. Figure 14 shows an array of choices among types of licenses for different types of components that appear in the OA case study example. Each choice determines the obligations that component producers can demand of their consumers in exchange for the access/usage rights they offer

The interdependence of producers, integrators, and consumers results in a co-evolution of software systems and social networks within an OA ecosystem [32]. Closely-coupled components from different producers must evolve in parallel in order for each to provide its services, as evolution in one will typically require a matching evolution in the other. Producers may manage their evolution with a loose coordination among releases, for example as between the Gnome and Mozilla organizations. Each release of a producer component create a tension

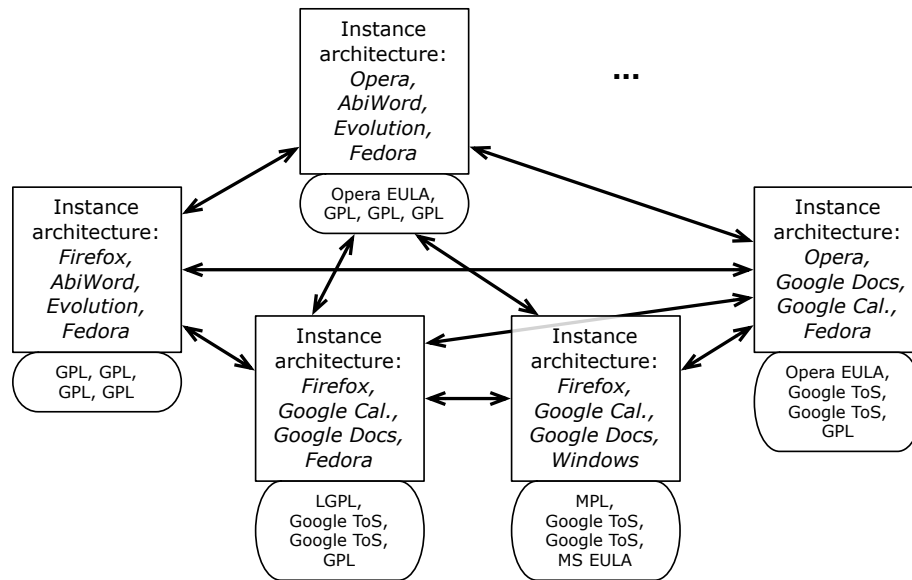


Figure 13: Possible evolutionary paths among a few instance architectures; some paths are impractical due to the changes in license obligations

	Browser	Word processor	Calendar, email	Platform
Proprietary	Opera (Opera EULA)	WordPerfect (Corel License)		Windows (MS EULA)
Strongly Reciprocal	Firefox (MPL or LGPL or GPL)	AbiWord (GPL)	Gnome Evolution (GPL)	Fedora (GPL)
Weakly Reciprocal or Academic		OpenOffice (LGPL)		FreeBSD (BSD variant)
Service		Google Docs (Google ToS)	Google Calendar (Google ToS)	

Figure 14: Some architecture choices and their license categories

through the ecosystem relationships with consumers and their releases of OA systems using those components, as integrators accommodate the choices of available, supported components with their own goals and needs. As discussed in our previous work [3], license rights and obligations are manifested at each component interface then mediated through the OA of the system to entail the rights and corresponding obligations for the system as a whole. As a result, integrators must frequently re-evaluate the OA system rights and obligations. In contrast to homogeneously-licensed systems, license change across versions is a characteristic of OA ecosystems, and architects of OA systems require tool support for managing the ongoing licensing changes.

6. Discussion

At least two topics merit discussion following from our approach to understanding of software ecosystems and ecosystem niches for OA systems: first, how might our results shed light on software systems whose architectures articulate a software product line; and second, what insights might we gain based on the results presented here on possible software license architectures for mobile computing ecosystems. Each is addressed in turn.

Software product lines (SPLs) rely on the development and use of explicit software architectures [6, 13]. However, the architecture of an SPL or software ecosystem does not necessarily require an OA—there is no need for it to be open. Thus, we are interested in discussing what happens when SPLs may conform to an OA, and to an OA that may be subject to heterogeneously licensed SPL components. Three considerations come to mind:

1. If the SPL is subject to a single homogeneous software license, which may often be the case when a single vendor or government contractor has developed the SPL, then the license may act to reinforce a vendor lock-in situation with its customers. One of the motivating factors for OA is the desire to avoid such lock-in, whether or not the SPL components have open or standards-compliant APIs. However, a single license simplifies determination of the software ecosystem in which these system is located.

2. If an OA system employs a reference architecture, then such a reference or design-time architecture effectively defines an SPL consisting of possible different system instantiations composed from similar components from different producers (e.g., different but equivalent Web browsers, word processors, calendaring and email applications). This can be seen in the design-time architecture depicted in Figure 5, the build-time architecture in Figure 6, and the instantiated run-time architectures in Figures 7, 8, and 9.
3. If the SPL is based on an OA that integrates software components from multiple producers or OSS components that are subject to different heterogeneous licenses, then we have the situation analogous to what we have presented in this paper, but now in the form of virtual SPLs from a virtual software production enterprise [26]. that spans multiple independent OSS projects and software production enterprises. As such, SPL concepts are compatible with OA systems that are composed from heterogeneously licensed components, but do not impact the formation or evolution of the software ecosystem niches where such systems may reside.

Our approach for using open software system architectures and component licenses as a lens that focuses attention to certain kinds of relationships within and across software supply networks, software ecosystems, and networks of software ecosystems has yet to be applied to systems on mobile computing platforms. Bosch [7] notes this is a neglected area of study, but one that may offer interesting opportunities for research and software product development. Thus, what happens when we consider Apple iPhone/iPad OS, Google Android OS phones, Nokia Symbian OS phones, Nokia Maemo OS hand-held computers, Microsoft Windows 7 OS phones, Intel Moblin OS netbooks, or Nintendo DS portable game consoles as possible platforms for OA system design and deployment? First, all of these devices are just personal computers with operating systems, albeit in small, easy to carry, and wireless form factors. They represent a mix of mostly proprietary operating system platforms, though some employ

Linux-based or other OSS alternative operating systems. Second, Mobile OS platforms owners (Apple, Nokia, Google, Microsoft) are all acting to control the software ecosystems for consumers of their devices through establishment of logically centralized (but possibly physically decentralized) application distribution repositories or online stores, where the mobile device must invoke a networked link to the repository to acquire (for fee/free) and install apps. Apple has had the greatest success in this strategy and dominates the global mobile application market and mobile computing software ecosystems. But overall, OA systems are not necessarily excluded from these markets or consumers. Third, given our design-time architecture of the case study system shown in Figure 5, is it possible to identify a build-time version that could produce a run-time version that could be deployed on most or all of these mobile devices? One such build-time architecture would compose an Opera Web browser, with Web services for word processing, calendaring and email, that could be hosted on either proprietary or OSS mobile operating systems. This alternative arises since Opera Software has produced run-time versions of its proprietary Web browser for these mobile operating systems, for accessing the Web via a wireless/cellular phone network connection. Similarly, in Figure 13 the instance architecture on the right could evolve to operate on a mobile platform like an Android-based mobile device or Symbian-based cell phone. So it appears that mobile computing devices do not pose any unusual challenges for our approach in terms of understanding their software ecosystems or the ecosystem niches for OA systems that could be hosted on such devices.

7. Conclusion

The role of software ecosystems in the development and evolution of heterogeneously-licensed open architecture systems has received insufficient consideration. Such systems are composed of components potentially under two or more licenses, open source software or proprietary or both, in an architecture in which evolution can occur by evolving existing components, replacing them, or refactoring. The software licenses of the components both facilitate and constrain

in which ecosystems a composed system may lie. In addition, the obligations and rights carried by the licenses are transmitted from the software component producers to system consumers through the architectural choices made by system integrators. Thus software component licenses help determine the contours of the software supply network and software ecosystem niche that emerge for a given implementation of a composed system architecture. Accordingly, we described examples for systems whose host software platform span the range of personal computer operating systems, Web services, and mobile computing devices.

Consequently, software component licenses and the architectural composition of a system determine the software ecosystem niche where a systems lies. Understanding and describing software ecosystem niches is a key contribution of this work. A case study of an open architecture software system that articulates different niches was employed to this end. We examined how the architecture and software component licenses of a composed system at design-time, build-time, and run-time helps determine the system’s software ecosystem niche, and provides insight for identifying potential evolutionary paths of software system, architecture, and niches. Similarly, we detailed the ways in which a composed system can evolve over time, and how a software system’s evolution can change or shift the software ecosystem niche in which the system resides and thus producer-consumers relationships. Then we described how virtual software product lines can be identified through a lens that examines the association between open architectures, software component licenses, and software ecosystems.

Finally, in related work [2, 3, 4] we identified structures for modeling software licenses and the license architecture of a system, and automated support for calculating its rights and obligations. Such capabilities are needed in order to manage and track an OA system’s evolution in the context of its ecosystem niche. We have outlined an approach for achieving these structures and support and sketched how they further the goal of reusing and exchanging alternative software components and software architectural compositions. More work re-

mains to be done, but we believe this approach transforms a vexing problem of stating in detail how study of software ecosystems can be tied to core issues in software engineering like software architecture, product lines, component-based reuse, license management, and evolution, into a manageable one for which workable solutions can be obtained.

Acknowledgments

This research is supported by grants #0534771 and #0808783 from the U.S. National Science Foundation, and the Acquisition Research Program at the Naval Postgraduate School. No review, approval, or endorsement is implied.

References

- [1] Alspaugh, T. A., Antón, A. I., 2008. Scenario support for effective requirements. *Information and Software Technology* 50 (3), 198–220.
- [2] Alspaugh, T. A., Asuncion, H. U., Scacchi, W., May 2009. Analyzing software licenses in open architecture software systems. In: 2nd International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS). pp. 1–4.
- [3] Alspaugh, T. A., Asuncion, H. U., Scacchi, W., 2009. Intellectual property rights requirements for heterogeneously-licensed systems. In: 17th IEEE International Requirements Engineering Conference (RE'09). pp. 24–33.
- [4] Alspaugh, T. A., Asuncion, H. U., Scacchi, W., 2009. The role of software licenses in open architecture ecosystems. In: First International Workshop on Software Ecosystems (IWSECO-2009). pp. 4–18.
- [5] Bass, L., Clements, P., Kazman, R., 2003. *Software Architecture in Practice*. Addison-Wesley Longman.
- [6] Bosch, J., 2000. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley.

- [7] Bosch, J., 2009. From software product lines to software ecosystems. In: 13th International Software Product Line Conference (SPLC'09). pp. 111–119.
- [8] Bosch, J., Bosch-Sijtsema, P., 2010. From integration to composition: On the impact of software product lines, global development and ecosystems. *J. Syst. Softw.* 83 (1), 67–76.
- [9] Boucharas, V., Jansen, S., Brinkkemper, S., 2009. Formalizing software ecosystem modeling. In: First International Workshop on Open Component Ecosystems (IWOCE'09). pp. 41–50.
- [10] Breaux, T. D., Anton, A. I., 2005. Analyzing goal semantics for rights, permissions, and obligations. In: RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering. pp. 177–188.
- [11] Breaux, T. D., Anton, A. I., 2008. Analyzing regulatory rules for privacy and security requirements. *IEEE Transactions on Software Engineering* 34 (1), 5–20.
- [12] Brown, A. W., Booch, G., 2002. Reusing open-source software and practices: The impact of open-source on commercial vendors. In: *Software Reuse: Methods, Techniques, and Tools (ICSR-7)*. pp. 381–428.
- [13] Clements, P., Northrop, L., 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional.
- [14] Dashofy, E. M., 2007. Supporting stakeholder-driven, multi-view software architecture modeling. Ph.D. thesis, University of California, Irvine.
- [15] Feldt, K., 2007. *Programming Firefox: Building Rich Internet Applications with XUL*. O'Reilly Media, Inc.
- [16] Firesmith, D., Jan.–Feb. 2004. Specifying reusable security requirements. *Journal of Object Technology* 3 (1), 61–75.

- [17] German, D. M., Hassan, A. E., 2009. License integration patterns: Dealing with licenses mismatches in component-based development. In: 28th International Conference on Software Engineering (ICSE '09). pp. 188–198.
- [18] Jansen, S., Brinkkemper, S., Finkelstein, A., 2009. Business network management as a survival strategy: A tale of two software ecosystems. In: First Workshop on Software Ecosystems. pp. 34–48.
- [19] Jansen, S., Finkelstein, A., Brinkkemper, S., 2009. A sense of community: A research agenda for software ecosystems. In: 28th International Conference on Software Engineering (ICSE '09), Companion Volume. pp. 187–190.
- [20] Jensen, C., Scacchi, W., Jul./Sep. 2005. Process modeling across the web information infrastructure. *Software Process: Improvement and Practice* 10 (3), 255–272.
- [21] Kuehnle, A.-K., Jun. 2008. Microsoft, open source and the software ecosystem: of predators and prey—the leopard can change its spots. *Information & Communication Technology Law* 17 (2), 107–124.
- [22] Kuhl, F., Weatherly, R., Dahmann, J., 1999. *Creating computer simulation systems: an introduction to the high level architecture*. Prentice Hall.
- [23] Messerschmitt, D. G., Szyperski, C., 2003. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. MIT Press.
- [24] Meyers, B. C., Oberndorf, P., 2001. *Managing Software Acquisition: Open Systems and COTS Products*. Addison-Wesley Professional.
- [25] Nelson, L., Churchill, E. F., 2006. Repurposing: Techniques for reuse and integration of interactive systems. In: *International Conference on Information Reuse and Integration (IRI-08)*. p. 490.
- [26] Noll, J., Scacchi, W., Feb. 1999. Supporting software development in virtual enterprises. *J. of Digital Information* 1 (4).

- [27] Noll, J., Scacchi, W., 2001. Specifying process-oriented hypertext for organizational computing. *J. Network and Computing Applications* 24 (1), 39–61.
- [28] Open Source Initiative, 2010. Open Source Definition. <http://www.opensource.org/docs/osd>.
- [29] Oreizy, P., 2000. Open architecture software: A flexible approach to decentralized software evolution. Ph.D. thesis, University of California, Irvine.
- [30] Ovaska, P., Rossi, M., Marttiin, P., 2003. Architecture as a coordination tool in multi-site software development. *Software Process: Improvement and Practice* 8 (4), 233–247.
- [31] Rosen, L., 2005. Open Source Licensing: Software Freedom and Intellectual Property Law. Prentice Hall.
- [32] Scacchi, W., 2007. Free/open source software development: Recent research results and emerging opportunities. In: 6th Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007). pp. 459–468.
- [33] Scacchi, W., Alspaugh, T. A., May 2008. Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In: 5th Annual Acquisition Research Symposium.
- [34] Taylor, R. N., Medvidovic, N., Dashofy, E. M., 2009. Software Architecture: Foundations, Theory, and Practice. Wiley.
- [35] Unity Technologies, Dec. 2008. End User License Agreement. <http://unity3d.com/unity/unity-end-user-license-2.x.html>.
- [36] van Gurp, J., Prehofer, C., Bosch, J., 2010. Comparing practices for reuse in integration-oriented software product lines and large open source software projects. *Software — Practice & Experience* 40 (4), 285–312.

- [37] Ven, K., Mannaert, H., 2008. Challenges and strategies in the use of open source software by independent software vendors. *Information and Software Technology* 50 (9-10), 991–1002.
- [38] Yau, S. S., Chen, Z., 2006. A framework for specifying and managing security requirements in collaborative systems. In: *Third International Conference on Autonomic and Trusted Computing (ATC 2006)*. pp. 500–510.

Recent Advances in Virtual Worlds for Science and Technology Research and Development

Walt Scacchi

Center for Computer Games and Virtual Worlds

Donald Bren School of Information and Computer Sciences

University of California, Irvine

<http://cgvw.ics.uci.edu>

Overview

- Recent virtual world projects for Science or Technology R&D
- Future opportunities for virtual worlds for science and technology R&D

Strategies for Creating Value with Virtual Worlds

- Creating game-based learning environments with virtual worlds
 - “Play” and experiential behavior are surprisingly effective way to audition, rehearse, act, fail, and learn
 - Mixed reality worlds can link virtual and physical activities
 - Virtual worlds are best at providing new *experiences*
 - Virtual work practices
 - Not the same as existing work practices
 - Need to learn what to do, how to do it, and more
 - Not obvious how to be faster, better, and cheaper using virtual worlds!

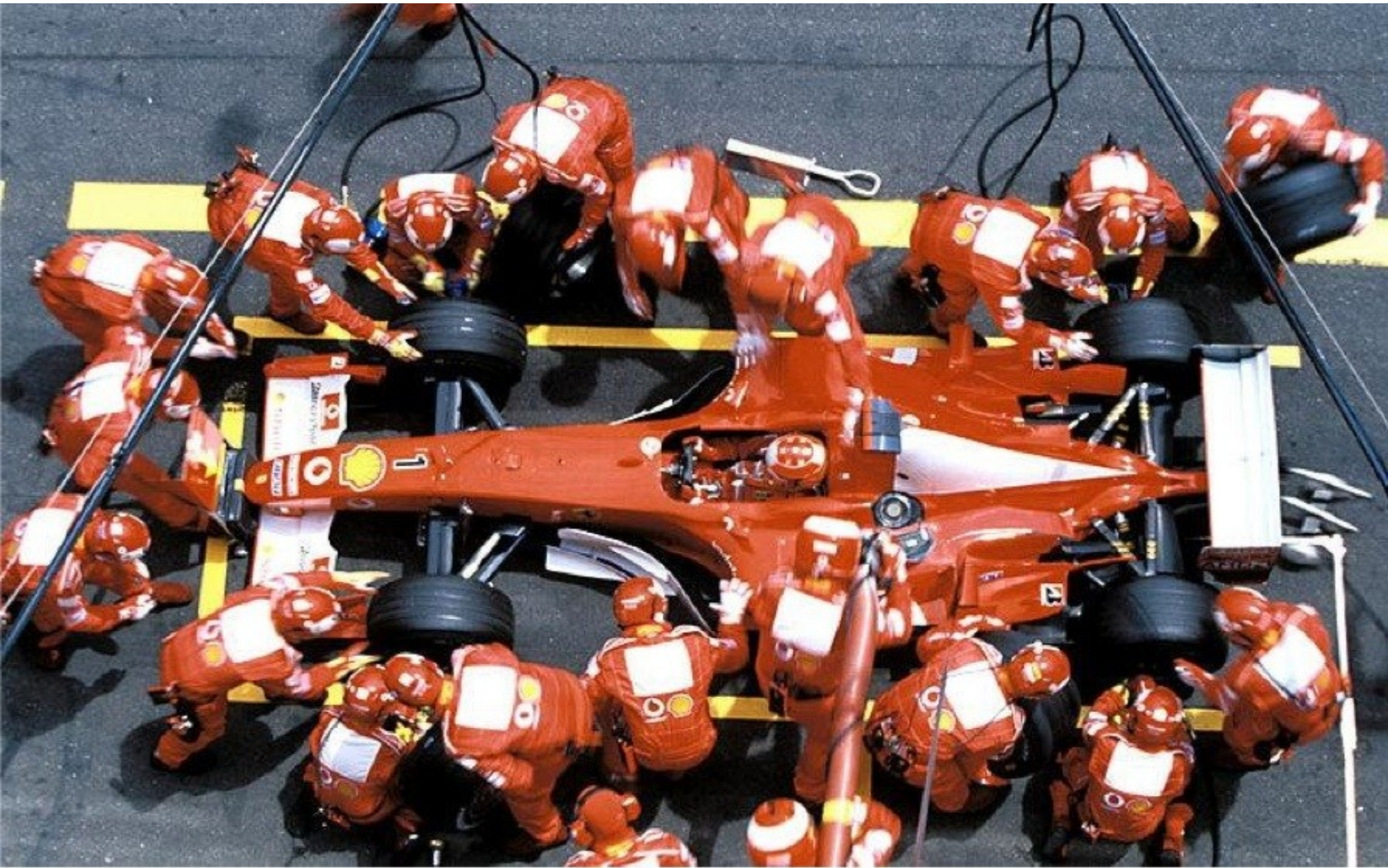
Collaborative meeting work in virtual world



Collaborative work in physical world



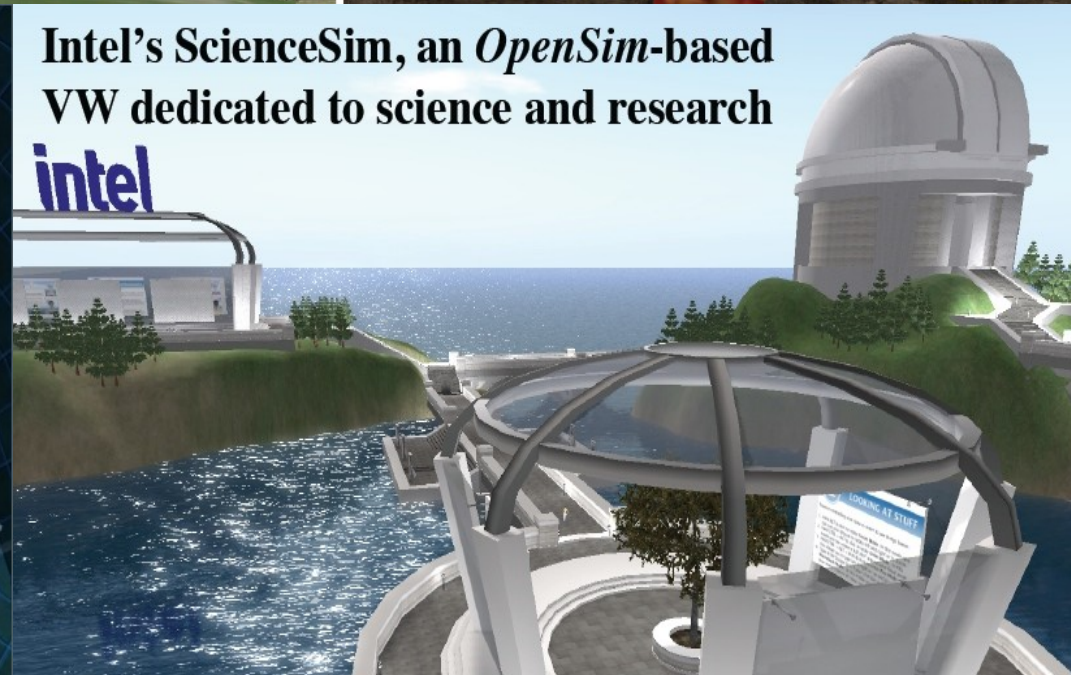
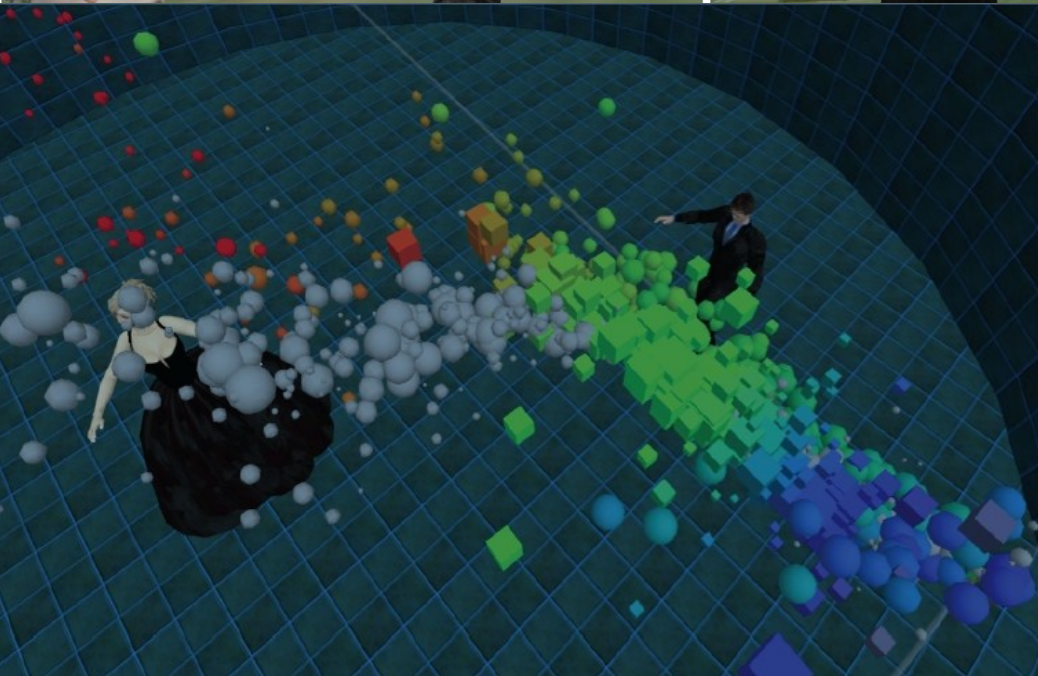
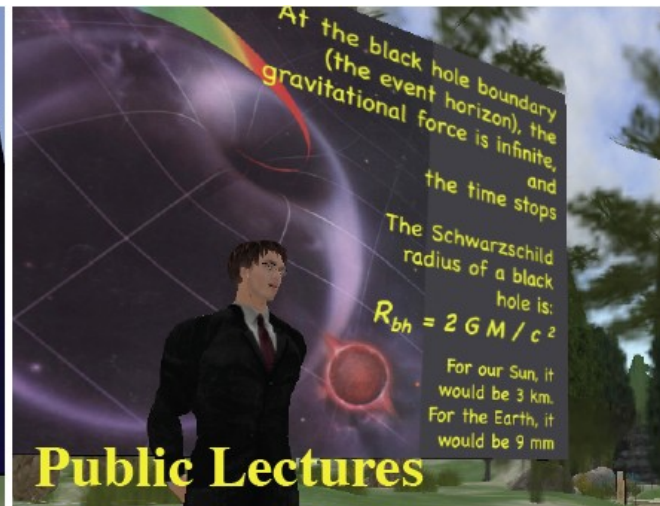
Radically colocated work in physical world



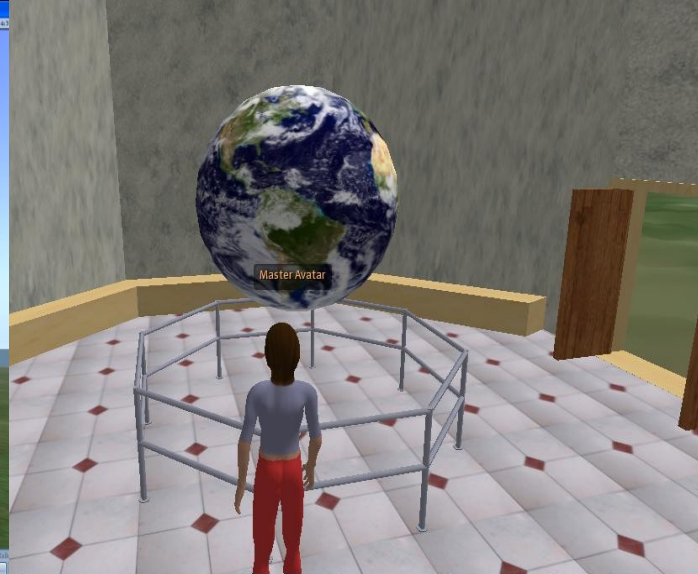
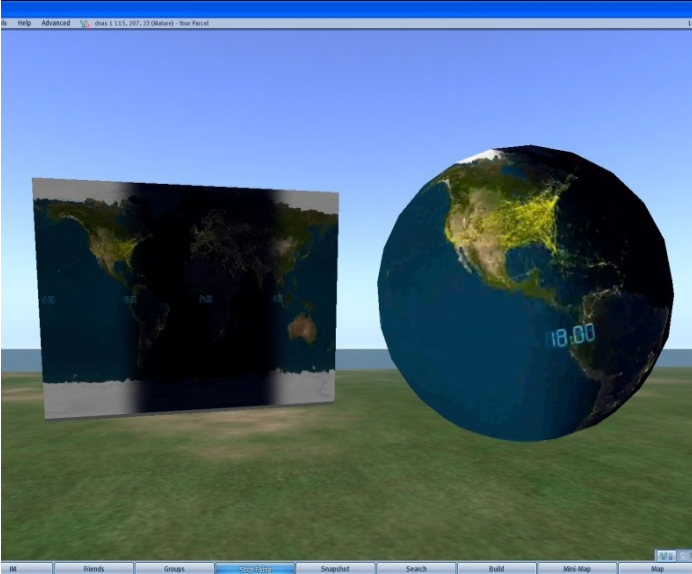
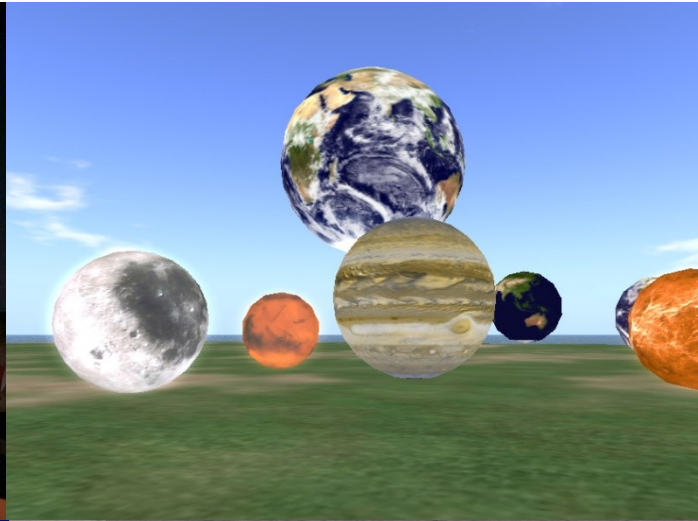
Recent Virtual World Projects for Science and Technology R&D

- Collaborative science meetings and immersive simulations
 - Meta Institute for Computational Astrophysics
- Collaborative science learning and data exploration environment with spherical displays at *Discovery Science Center* and in *OpenSim*
 - Science on a Sphere
- Collaborative game world for semiconductor fabrication or nanotechnology design
 - FabLab training simulator
- Game-based virtual worlds for advanced health care
 - Robotic therapeutics and tele-rehabilitation
- Envisioning future virtual worlds for possible cultural experiences and new technological innovation opportunities
 - Virtual Life 2010+
 - Immersive motorsports racing experiences
 - Low-cost to high-cost virtual world simulators
 - OutRun @ UCI

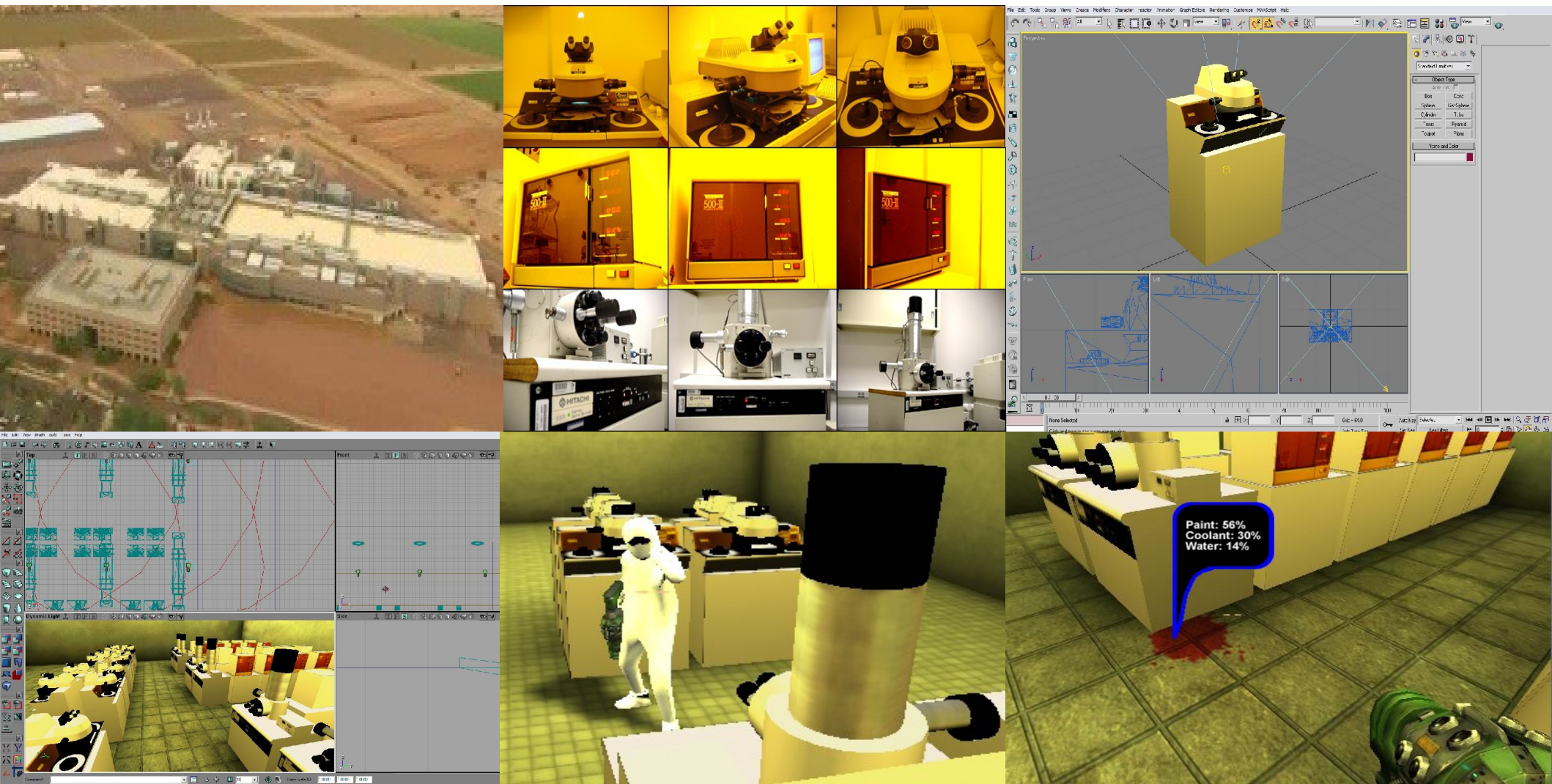
Virtual Worlds for Scientific Collaboration: *Meta Institute for Computational Astrophysics*



Spherical displays and “spherescasting” support: *NOAA Science on a Sphere* installation in *Opensim*



Game-based virtual world for semiconductor/nanotech fabrication training, remote presence and diagnosis



FabLab Demo Reel

Semiconductor/nanotechnology fabrication training game

working in a cleanroom

Suit made of
ultra clean material

Battery pack for
air filter system

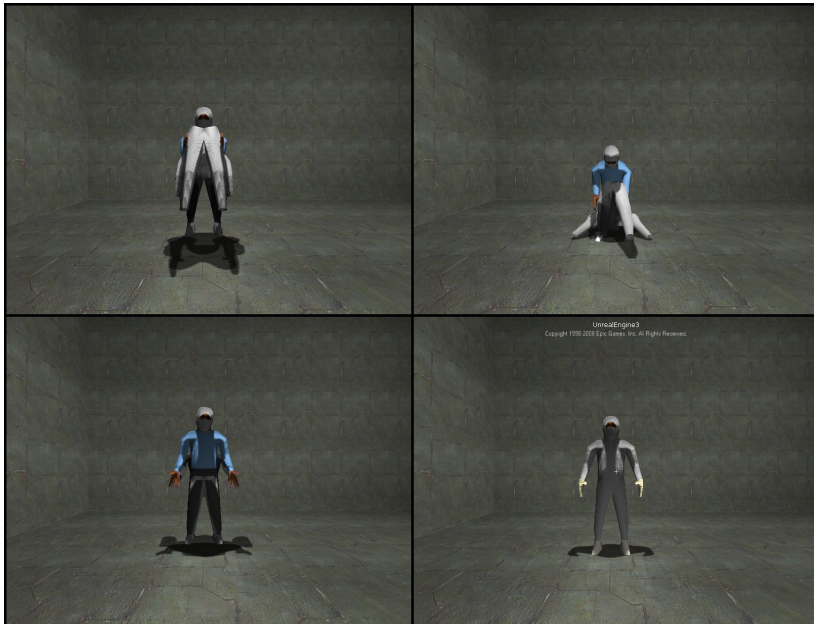
2 pairs of gloves
nylon & latex

2 pieces
of foot gear
disposable
shoe covers &
outer booties

Helmet
includes
air filter
unit

Will also
wear
hairnet
& safety
glasses

Belt



Virtual worlds for health care and tele-rehabilitation

- **Virtual worlds can be used to support various kinds of tele-medicine and tele-robotics applications/tasks**



- **“Rehabilitation” tasks supported can include:**
 - **Remote observation, tele-consultation, role-playing and identity switching through avatars, device data collection, device software updates, collaborative product/prototype development, and more**

Assisted performance training and robotic rehabilitation

- **Wii Sports** (best selling game for Nintendo in 2007; 45M copies sold worldwide through 2009)

- **Boxing**
- **Bowling**
- **Golf**
- **Tennis**
- **Baseball**



What's next?



VW haptic interfaces with therapeutic applications

- Simulated devices
 - Guitar Hero guitar; Rock Band drum set
- Haptic wheels, trackballs, and joysticks
- Force-feedback play controllers (racing game wheels, pneumatic bladders)
- Multi-sensor play controllers (including video capture, infra-red, accelerometers, neurological sensors, electro-goniometers (SEMG), etc.)
 - Wii Remote and nunchuk
- Multi-jointed, body-worn sensors as play controllers
 - Data gloves



- GypsyMIDI



Haptic interfaces with possible therapeutic applications

- Endoscopic surgery training “joysticks”

- Simball 4D joystick adapted to therapeutic game play for stroke rehabilitation

- <http://www.g-coder.com/content/view/7/6/>



- 3D, real-time video motion capture enabling *mixed reality game play* spanning physical and virtual worlds

- *Project Natal* at Microsoft
- In-game characters can interact with human players through gestures and body movements
- http://www.youtube.com/watch?v=g_txF7iETX0



Some findings on Games for Health/Therapeutic Applications

- The design and utility of a game to realize therapeutic value is not obvious.

E. Flores, G. Tobon, et al.,
Improving Patient Motivation in
Game Development for Motor
Deficit Rehabilitation, *ACM 2008
Intern. Conf. Advances in Computer
Entertainment*, 381-384.

Table 1. Gaming design criteria for stroke rehabilitation programs serving elderly users

Criteria for Stroke Rehabilitation	Criteria for Elderly Entertainment
- Adaptability to motor skill level - Meaningful tasks - Appropriate feedback - Therapy-Appropriate ROM - Focus diverted from exercise	- Appropriate cognitive challenge - Simple objective/interface - Motivational Feedback - Element of social activity - Appropriateness of genre - Creation of new learning following guidelines of experts - Sensitivity to decreased sensory acuity and slower responses

		Pong	Driver's SEAT	Whack-a-mouse	Tetris	Computer Chess	Trivial Pursuit
CRITERIA	Stroke Rehab	Adaptability to motor skill level	✓	✓	✓		
		Meaningful tasks	✓	✓			
		Appropriate feedback		✓			
		Therapy-appropriate ROM		✓			
		Focus diverted from exercise	✓	✓	✓	✓	✓
	Elderly Entertainment	Appropriate cognitive challenge			✓	✓	✓
		Simple objective/interface	✓	✓	✓	✓	✓
		Motivational Feedback	✓	✓	✓	✓	✓
		Element of social activity	✓			✓	✓
		Appropriateness of genre	✓	✓	✓	✓	✓
		Creation of new learning				✓	✓
		Sensitivity to decreased sensory acuity	✓	✓	✓	✓	✓
		Sensitivity to slower responses	✓	✓	✓	✓	✓

Envisioning collaborative virtual worlds 2010-2012



Virtual Life Demo Reel

Game-Based Virtual World Simulator Interfaces for immersive motorsports racing experiences



Game-based virtual world simulator you can actually drive in physical world! -- *OutRun* @ UCI



Future opportunities for games and virtual worlds

- Key challenges to address/overcome -- *scale and scope of:*
 - *Immersion*
 - *Verisimilitude*
 - *Within worlds*
 - *Spanning physical-virtual worlds*
 - *Co-participation and Collaborative work*
 - *Relocatability (telepresence)*
 - *Decentralized virtual organization*
- *New research center for Computer Games and Virtual Worlds at UCI*
 - <http://cgvw.ics.uci.edu>
 - *Funding from National Science Foundation #0808783, Digital Industry Promotion Agency (Daegu, South Korea), and others.*
 - *Want to come and play with us?*