Amethyst: A 3D Adventure Game for Tablets

Ross Light
California Polytechnic State University
Advisor: Dr. Zoë Wood

March 20, 2013

Abstract

Amethyst is a point-and-click science fiction adventure game for tablets, developed over two quarters to demonstrate applied principles of computer graphics and software engineering discipline. Using a novel interaction system, players can engage in environmental puzzles. Development of Amethyst required non-trivial quality assurance measures and workflow enhancements. The end result is a functional technical demo showcasing a single level, with the ability to rapidly produce more content.
1 Introduction

The process of developing a modern video game utilizes many disparate fields of study in computer science. For example, real-time 3D rendering takes advantage of parallel processing and operating system interactions. While the act of producing low-level game engine technologies is a well-recognized field of study, the ability to provide a compelling experience via interaction between a game and a player is an under-appreciated art. Amethyst is an attempt to study both the potential of interactive storytelling and the process necessary to support a game development team.

Amethyst is a point-and-click science fiction adventure game for tablets, developed by Ross Light and Adam Hintz over two quarters (essentially six months), with the narrative written in the three months before production. In the story, a large unnamed corporation loses contact with Amethyst, a ship designed to transport a group of scientists to examine an artifact found on the surface of Mars. The player character’s goal is to discover the source of the problem and repair it. Through exploring the environment and solving puzzles, the player realizes that the Amethyst’s autonomous system has become self-aware, calling itself Raven. The story concludes with a decision the player must make: the player can either allow Raven to go to Mars by herself or the player can take control of the ship and return Raven to Earth for study.

During the two quarters of production, Adam and Ross developed a formal “pipeline” for creating digital art assets, a novel environmental interaction system built on top of the commercial Unity game engine, and a functional first level of the game.


2 Related Work

There are countless commercial video games that influenced both the gameplay and narrative of Amethyst. The most significant direct influences on narrative are Mass Effect, Myst, and Portal. The game mechanics of Amethyst are most affected by Myst and Penumbra: Overture.

2.1 Mass Effect

Mass Effect, a science fiction role-playing game developed by BioWare in 2007 (shown in Figure 1), influenced Amethyst’s narrative and themes. Mass Effect’s character interactions are based around a moral choice system with a divergent narrative that allows decisions made in an earlier game to affect events in later sequels. Recurring themes in the Mass Effect series include xenophobia, the role of technology in society, and free will.

Figure 1: Mass Effect
2.2 Myst

Myst is a best-selling fantasy adventure game created by Cyan in 1993. The game and its sequels are notable for navigation based on clicking screen regions, limited player death, and pre-rendered visuals (one of which is shown in Figure 2). Myst V — the final installment in the series — is notable for defying the third characteristic by rendering its environments in real time, and optionally allowing the player to move without clicking screen regions.

The mechanics of movement in Amethyst build on the conventions introduced in Myst while adding more gestures that are applicable in the context of a touch interface.

2.3 Penumbra: Overture

In 2007, Frictional Games released Penumbra: Overture, a horror adventure game. The game is rendered in real-time, uses physics-based puzzles, and features an inventory system.
Figure 3 shows a scene early in the game that showcases the physics engine.

Penumbra: Overture, while having good puzzles, provided an example of gameplay mechanics that Amethyst specifically avoided: an inventory system and a complex gesture system. Both of the developers of Amethyst found these frustrating during play.

2.4 Other Influences

Many other video games influenced Amethyst in much smaller ways. For example:

• BioShock: A variant on the pipe puzzle was used for the Communication Room level, which was not completed for the demo.

• Portal: The tone, narrative, and characters from Portal influenced the narrative.

• Spec Ops: The Line: The style of having an opening menu that leads into the opening cutscene was directly taken from Spec Ops: The Line.
Machinarium: While not a direct influence on gameplay, it is a good example of a compelling input scheme that works well on tablets.

3 Why Unity?

Amethyst uses Unity as its game engine. Unity is a multi-platform rendering engine for creating interactive 3D content [4]. It features an extensible visual editor that emphasizes rapid prototyping and iteration between editing and testing. By including the Mono runtime, Unity provides a platform-agnostic execution environment that can be scripted in C#, Boo, or UnityScript (a proprietary language that is similar to JavaScript) [6].

The two targeted tablet-based operating systems (iOS and Android) have fundamentally different programming environments (Objective-C and Java, respectively). Because Unity uses the Mono runtime to ensure a consistent environment, Amethyst can be published simultaneously to both Android and iOS without compromising the performance or quality of the game on either platform [5]. This reduces the technical burden of creating a game engine from scratch, while retaining the control to script and develop features in a powerful language. The end effect is a focus during development time on iterative development of gameplay features.

Unity’s standard library includes not only vector manipulation and rendering routines, but pre-built assets and shaders. For instance, the trees in the Beach level were generated using a built-in procedural tree generation tool [7]. This packaging scheme (known as “prefabbing”) is available for user-created assets, which makes instancing inside a scene efficient.

Instead of a traditional inheritance-based idiom, Unity bases its scripting around composition [1, p. 20]. Individual scripts are “behaviors” that can be attached to game objects. These components can be built-ins (e.g. transform, mesh rendering) or user-defined (any
Mono script). Amethyst uses this to decouple movement logic from specific game/scripting logic.

4 Implementation Details

4.1 Continuous Integration

From the author’s previous experiences with large software projects, continuous integration improved code quality. Part of the convenience of a continuous integration system is the ability to check whether a particular revision compiles and passes all unit tests. Unity does not provide a unit testing framework, but does provide a batch command-line mode of operation.

The unit testing framework created for Amethyst provides non-terminating error messages and elides assert methods in favor of explicit conditionals, similar to the Go testing package [2]. The technical demo’s unit tests only verify low-level viewpoint math, but it is possible to mock the larger puzzle controllers. These tests can either be run interactively — via a custom Unity user interface — or from the Unity batch command-line mode.

To provide automated builds, a program was written to periodically pull changesets via Mercurial from each of the developers to a local build directory, then invoke Unity to compile the scripts and execute the unit tests in batch mode. The results would be pushed to a custom Google App Engine instance to be displayed in a dashboard with the most recent builds (shown in Figure 4). This architecture had the benefit of allowing an arbitrary number of build machines that could possibly fail or go offline while still maintaining a high-reliability dashboard.
4.2 Lock Server

The Amethyst team used Mercurial as its version control system. Mercurial is a decentralized version control system that has powerful ancestor merging [3], but no concept of file-locking. Since many of the digital art assets are large binary files that can’t be meaningfully sent to a three-way merge tool, the team needed a means of preventing unintended binary merge conflicts, as the consequences could mean hours of lost work.

A Mercurial plugin was written for Amethyst that added three commands: lock, unlock, and locks. The lock and unlock operations send HTTP requests to a known server address to publish information about which files the user is using. Additional metadata is recorded and displayed for each lock, such as creation date, author, and an optional comment. The files with locks currently held can be displayed with the locks operation. Upon every commit, the lock server is consulted to ensure that no other user holds a lock on a modified file, and aborts if this condition is not met. Additionally, the hg status command also
warns if a file is modified and another user holds a lock.

The lock server keeps a central datastore for locks currently held. This is implemented using transactions in the datastore of the same Google App Engine instance mentioned in the previous section. The lock and unlock requests are atomic to provide mutual exclusion. Using App Engine ensures high reliability and convenience (the dashboard also displays all currently locked files).

One of the potential races in this approach was releasing a lock and having another user not pull the affected changesets. However, during Amethyst’s development, this issue was prevented by clear communication and a strict discipline of pulling changesets before a work session.

4.3 Viewpoint System

The most notable characteristic of Amethyst (from the player’s perspective) is the navigation and interaction system. Much like Myst games after Myst III, the player can stand in a single point and get a panoramic view of their location by dragging around the screen (see Figure 5). Each of these positions are called viewpoints inside the Amethyst engine. A single tap will move the player to another viewpoint; environmental objects can be interacted with via taps and drags.

In designing the viewpoint system, the gestures needed to be simple to explain, the system needed to be able to disambiguate between gestures meant for looking and gestures meant for interacting, and the viewpoints needed to handle arbitrary up vectors gracefully (because the direction of gravity changes depending on where the player is in the station).

The gestures for interactions are explained in-game via a series of messages that appear on screen. Before these messages were added, players had a difficult time inferring what they needed to do to play the game. However, players found that once they were acquainted
with the controls, movement was simple and straightforward.

Disambiguation of gestures is done by casting a ray in the direction of the initial touch and performing box-ray collisions on objects (the internals of the collision code are handled by Unity). If the ray collides with an object, the gesture is assumed to mean an environmental interaction and the handling is passed to a script attached to the object. Otherwise, it is assumed to be either a look-drag or a move-tap.

To handle look-drags or move-taps, a viewpoint computes a spherical coordinate system whose pole is an up vector determined from the rotation of the game object (which defaults to positive Y). Yaw and pitch are defined relative to the up vector: yaw is rotation about the up vector and pitch is rotation about a vector orthogonal to both the up vector and vector being considered. The connections from a viewpoint to other viewpoints form a directed graph, where each edge is called a region. A region stores the yaw size, yaw offset, pitch size, and pitch offset in addition to the viewpoint it is connected to. These four numbers define a portion of the coordinate system’s spherical surface — relative to the normalized vector difference of the two viewpoints — that is used for hit tests when performing a move-tap. A look-drag computes the change in yaw and change in pitch and rotates the camera accordingly. A simple example of this coordinate system is shown in Figure 5.

To compute whether or not a tap is part of region, Amethyst follows this procedure: Let \( \vec{v} \) be the normalized vector in world space in the direction of the tap. Let \( \vec{u} \) be the normalized up vector in world space. Let \( \vec{x} = ||\vec{x}_2 - \vec{x}_1|| \), where \( \vec{x}_1 \) is the position in world space of the current viewpoint and \( \vec{x}_2 \) is the position in world space of the viewpoint being considered. The yaw offset, the yaw size, the pitch size, and pitch offset are written as \( \theta_y, \phi_y, \theta_p, \) and \( \phi_p \), respectively. Let \( \alpha = \frac{\pi}{2} - \arccos (\vec{u} \cdot \vec{x}) + \theta_p \) and \( \beta = \frac{\pi}{2} - \arccos (\vec{u} \cdot \vec{v}). \) Let \( \gamma = |\arccos [(\vec{x} \times \vec{u}) \cdot (\vec{v} \times \vec{u})] - \theta_y| \) if \( \vec{u} \cdot [(\vec{x} \times \vec{u}) \times (\vec{v} \times \vec{u})] \geq 0 \), otherwise let \( \gamma = \)
Figure 5: A single viewpoint during look-drag. $\theta_y = 0$ and $\theta_p = 0$.

$| - \arccos [(\vec{x} \times \vec{u}) \cdot (\vec{v} \times \vec{u})] - \theta_y |$. A tap is part of a region if one of the following is true:

1. $\gamma \leq \frac{\phi_y}{2}$ and $|\alpha - \beta| \leq \frac{\phi_p}{2}$

2. $\pi - \gamma \leq \frac{\phi_y}{2}$ and $\alpha + \frac{\phi_y}{2} > \frac{\pi}{2}$ and $\beta > \pi - (\alpha + \frac{\phi_y}{2})$

3. $\pi - \gamma \leq \frac{\phi_y}{2}$ and $\alpha - \frac{\phi_y}{2} < -\frac{\pi}{2}$ and $\beta < -\pi - (\alpha - \frac{\phi_y}{2})$

Figure 6: Wireframe top view of the Beach level after final layout
Because the viewpoints' target vectors are based on the vector between the two points, it is simple for a layout designer to iteratively test tweaks in the positioning of viewpoints during the layout phase (see Adam’s paper for details). The viewpoints can be used to direct the player toward important parts of a level and to allow the level designer to have finer control over the player’s experience, as can be seen in the layout shown in Figure 6.

4.4 Persistence

Amethyst is designed to have a simple key-value data store for saving and loading player progress without explicit interactions. This allows for a single saved game (which is sufficient for mobile games) and loose coupling of the save functionality and the game state. Each puzzle controller is responsible for handling its persistence.

Because Amethyst saves frequently and without direct user input, it is important that the scheme is fast and lightweight. To alleviate disk hits, Amethyst doesn’t immediately write changes to the key-value store; Amethyst requires an explicit commit. The commit operation is informed whether this is a “world-changing” commit or a “mundane” commit. A “world-changing” commit, such as progress in a puzzle or occurrence of a plot event, is written to disk immediately since having to redo effort would be disrupting to the player. More transient events (e.g. moving around and changing viewing angle) are deemed “mundane”, and it is up to the key-value store whether to write to disk. The key-value store may choose to queue events, maintain a rolling log, etc.

4.5 Introductory Cutscene

The introductory cutscene in Amethyst delivers narrative exposition to the player: his/her objective in the game, the scenario, and a small glimpse into the world. When the player taps “New Game”, the main menu (pictured in Figure 7) fades away and becomes the
establishing shot of the Amethyst. After a few lines, the camera cuts into the shuttle’s interior (pictured in Figure 8), allowing the player to start interacting with the game while the voiceover is still playing. This scene then leads into the first level of the game.

This scene was created in a previsualization form as a final project for CSC 474: Computer Animation. It was modeled, textured, and animated in Blender then presented for that class using a custom real-time rendering engine. However, additional work needed to be done to have the cutscene fit into the interactions inside Amethyst. The original Blender file remained unmodified (since Unity can import animations from Blender), but the animation was split at the camera cut, and another animation was created to isolate the rotation of the Amethyst. These two steps were performed during the import into Unity.

Using these pieces, the script controlling the introductory cutscene could rotate the Amethyst while the player is making his/her choice from the main menu. The menu fades out and the shuttle’s animation begins at the same time the voiceover begins playing. When the first shot’s animation finishes, the script changes the camera’s position into the shuttle, enables the viewpoint system script, then begins playing the second part of the animation for both the shuttle and the Amethyst. Even though this may cause a discontinuity in the Amethyst’s rotation, the discrepancy is obscured by the camera cut. Once the second animation clip is finished, the cutscene and the voiceover is complete, so the next level is loaded.

5 Results

At the end of the two quarters, Amethyst has one full level visually complete, a main menu (Figure 7) with an introductory cutscene, and ending credits. The game has been tested on a Samsung Galaxy Nexus, a Nexus 7, and a Lenovo Thinkpad Tablet (all Android
The game features tutorials for gameplay (Figure 8), a functioning puzzle (Figure 9, Figure 10, Figure 11), level transitions, and a scripted sequence (Figure 7, Figure 8). The persistence infrastructure was implemented during second quarter, and although the “world-changing” commits are recorded, there is no means for a user to perform a load. The viewpoints don’t have a unique naming scheme, so the level would need to be modified to support location storage.

The other levels (the Hub, the Nest, the Science Room, and the Communications Room) have been modeled and set up for “rough layout”, which means that they can be navigated using the viewpoint system. Sound effects, music, and dialogue are largely missing, but temporary assets are in place to prove that these systems work.

The pipeline created for Amethyst fulfilled its requirements. As predicted, the machine that ran the continuous integration server was not always online, but the dashboard page never had downtime. The lock server fulfilled its purpose, as there were no binary
merge conflicts during Amethyst’s development. Playtesters had no complaints about the viewpoints and interaction system, other than viewpoint placement in some cases.

5.1 Conclusion

Unity saved time by providing asset importers for 3D model formats and a rapid build-run-test cycle. Even though it lacked unit testing and version control (at least in the non-Pro version), it is extensible enough to create a powerful custom workflow integrated with the editor. The learning curve for basic features is reasonable, but trying to integrate it in automated tests requires more intimate knowledge of the entire engine. Unity exposes enough functionality to write a complex, novel interaction system without significant hurdles. By structuring scripts using composition instead of inheritance, complex scripting behaviors can be created with little effort. Overall, the experience with Unity was positive; the only qualms were based on ignorance of the engine. Unity allowed more development time to
be spent on gameplay iterations. We would recommend Unity for development of future game projects.
Figure 10: First Level — The Beach

Figure 11: First Level — The Beach
6 Future Work

Amethyst still has levels that are written but not finished in its repository. These levels need surface painting, lighting, sound effects, and final layout. When the game is complete, Adam and Ross plan on publishing Amethyst to the iOS App Store and Google Play. The iOS executable would need to be tested, although much of the portability is already provided by Unity.

The persistence infrastructure — although sound — still requires support from existing levels in order to create seamless transitions between levels and a “continue game” option. Viewpoints should be renamed to have unique names such that a load operation causes the player to move back to his or her previous viewpoint. This is probably the largest change to the fundamental gameplay architecture that remains to be completed. There are still some issues on Android devices with sleeping during cutscenes, which remain to be resolved, along with a few display hiccups (although these seem to be problems with the geometry, not a flaw with Unity itself).

The Amethyst team expects to continue working on this project in the coming months. The sound design has no technical issues, but the team was not able to secure high-quality audio assets before time of completion. The digital props are deemed sufficient, but of course, more details can be added. Viewpoints in existing levels may still need to be tweaked to provide a better user experience. Overall, the current state of the project is a functioning technical demo, but additional time is needed to provide a more compelling narrative.
References


