The Ground Is Lava!

Aaron Jacobs

California Polytechnic State University
Adviser: Professor Zoë Wood

June 2015

Contents

1

Abstract

The Ground Is Lava! is a three dimensional video game written in C++ that uses OpenGL as its graphics API. The game is competitive, with two to four players controlling characters from a first-person perspective. The project implements multiple graphics technologies in order to achieve a consistent, pleasing visual style, including shadow mapping, sky rendering, and procedural animation. The engine built to power the game was developed in a flexible manner, allowing the code to be reused for future projects.

2

Introduction

Video games have grown significantly since their early days. More and more, video games are becoming respected as a medium. In 2012, the Smithsonian even held an exhibit named "The Art of Video Games" [Smithsonian Institution, 2012]. From a computer science perspective, video games are complex software systems encompassing many topics, including physics, graphics, audio, and more. Their complicated nature makes them a prime choice for projects, as they provide an opportunity to express a thorough understanding of software development.

2.1 Goals

At the start of the project, there were three main goals. First, to create an experience that would be fun (putting focus not only on the technical aspects of the project, but on design and art direction as well). Second, to write a custom engine (C++ / OpenGL) in order to gain more insight into the game programming process. Finally, to get the game to some state of completion, so that it could be submitted to IndieCade (an independent game developers' conference) before graduation.

It was known from the outset that development time would be very limited (about twenty weeks). Therefore, a game genre was chosen that would allow the project's scope to be limited.

2.2 Inspiration

Fond memories exist from younger years of spending the night at friends' houses on weekends, staying up late, playing competitive multiplayer games. While online play has done a lot to evolve the multiplayer experience, it seems that there is something special about sitting around a screen and yelling obscenities at friends in person.

The desire to recreate this experience acted as the motivation for creating The Ground Is Lava! - a relatively simple competitive game that incites a competitive spirit between friends.

3

Project Overview

3.1 Description

The Ground Is Lava! is a local / splitscreen multiplayer first-person party action platformer. The game takes place on a series of islands floating above a sea of lava. Each player controls a golem with the ability to throw explosive rocks and push other players. The powerful golems are unable to be hurt by each others' attacks, being susceptible only to the lava that lies below. The goal is simple: throw your opponents to their fiery demise and be the last one standing. Games consist of a series of short rounds, with a winner declared once a player has achieved a set number of victories. The game is simple to learn, and was designed to be played at parties.

3.2 Development

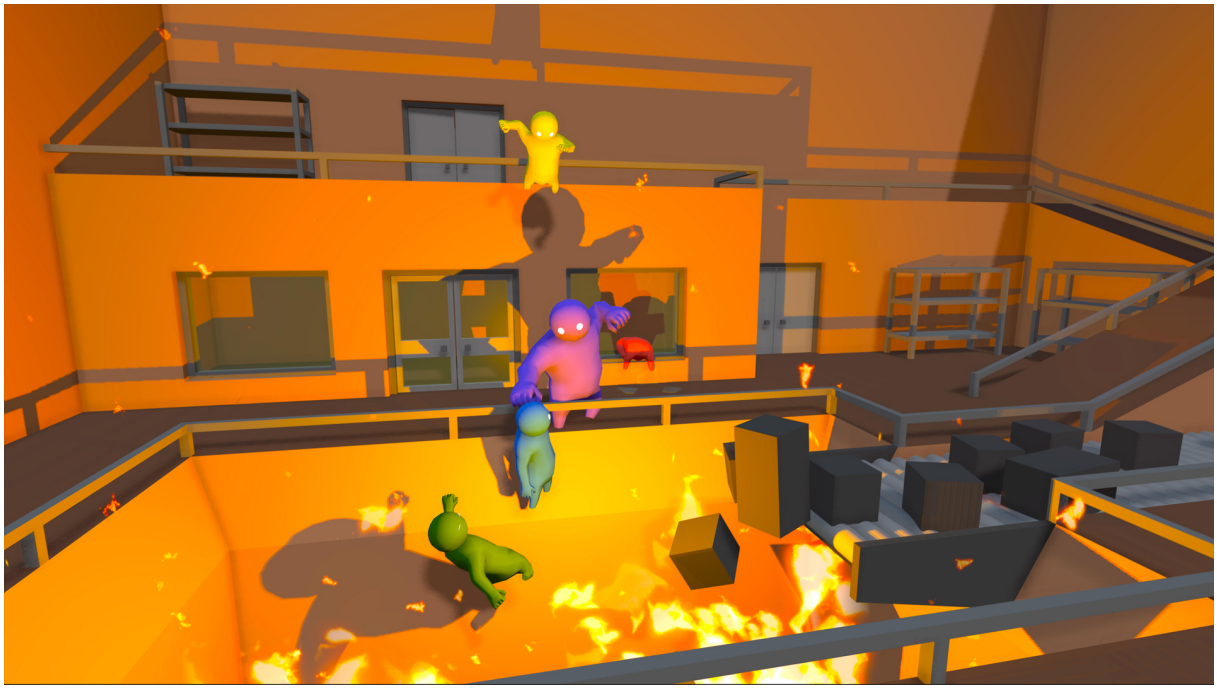
The project was developed over the course of twenty weeks by a single person. It was written in C++, using OpenGL for its graphics API (targeting version 3.3). Cross-platform compatibility was a goal from the start of the project, and the game is able to run on Windows, OS X, and Linux. Multiple libraries were used in development, including the Open Asset Import Library (model importing) [Assimp Development Team, 2014], Bullet (physics) [Real-Time Physics Simulation, 2015], FMOD (audio) [Firelight Technologies, 2015], GLFW (windowing and input) [The GLFW Development Team, 2015], glm (math) [G-Truc Creation, 2015], and stb (image and font loading) [Barrett, 2015].

4

Related Work

The Ground Is Lava! was heavily inspired by games played at parties around GDC. The three following titles had some of the greatest influence.

4.1 Gang Beasts



Boneloaf's Gang Beasts [Boneloaf, 2014] was a huge inspiration for The Ground Is Lava!. Its gameplay is very simplistic, only giving players the ability to move, jump, flail their arms, and grab. The fun and excitement of the game are driven by the interactions between the players themselves.

Gang Beasts' geometrically minimalistic, but very colorful art style influenced the art style of The Ground Is Lava!.

4.2 TowerFall Ascension

TowerFall Ascension [Matt Makes Games, 2014] is an incredibly addictive fast-paced game. It has a multiplayer mode where four people square off to be the last one standing. The game's speed and highly competitive nature greatly influenced the design of The Ground Is Lava!

4.3 Nidhogg

Nidhogg's [Messhof, 2014] simple premise allows for it to be picked up easily. It is impressive how fast newcomers to the game are able to get started. Its simplicity encouraged The Ground Is Lava! to be created as a game that is capable of generating a large amount of excitement in little time.

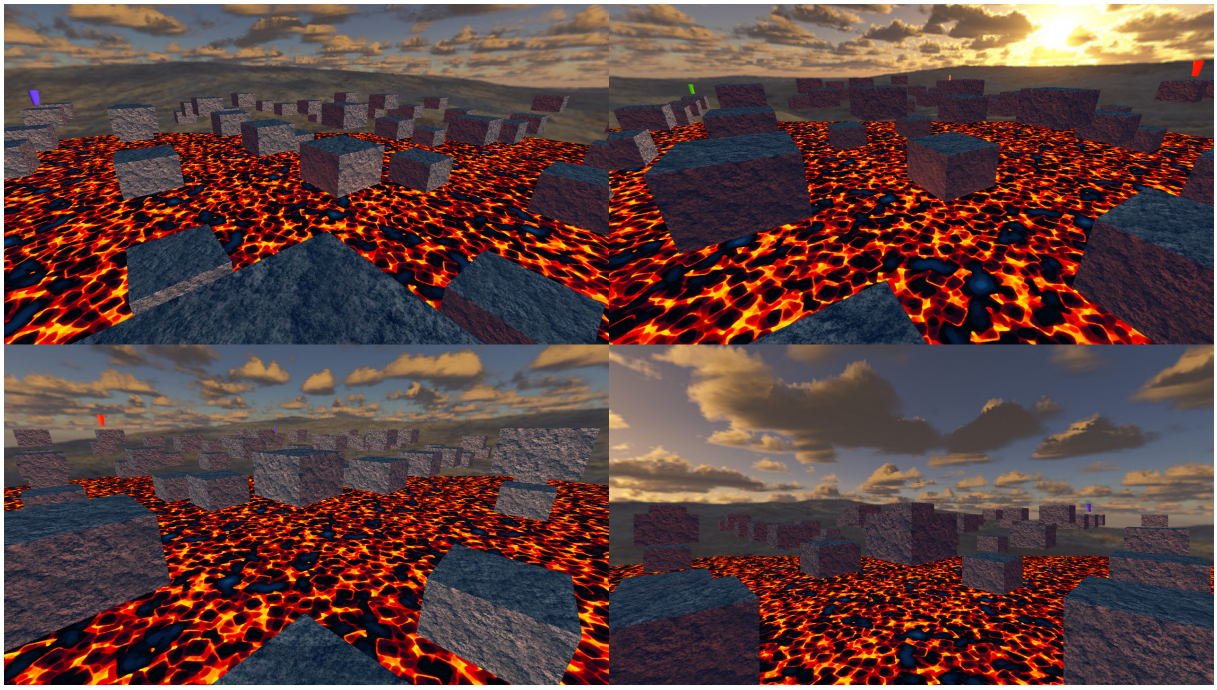
5

Feature Timeline

The production timeline for the project was short; therefore, features were broken down into groups. These groups helped guide the development process by encouraging gameplay-critical features to be completed early, in order to facilitate playtesting. Having early playtests was crucial, as it provided necessary feedback for the game development process.

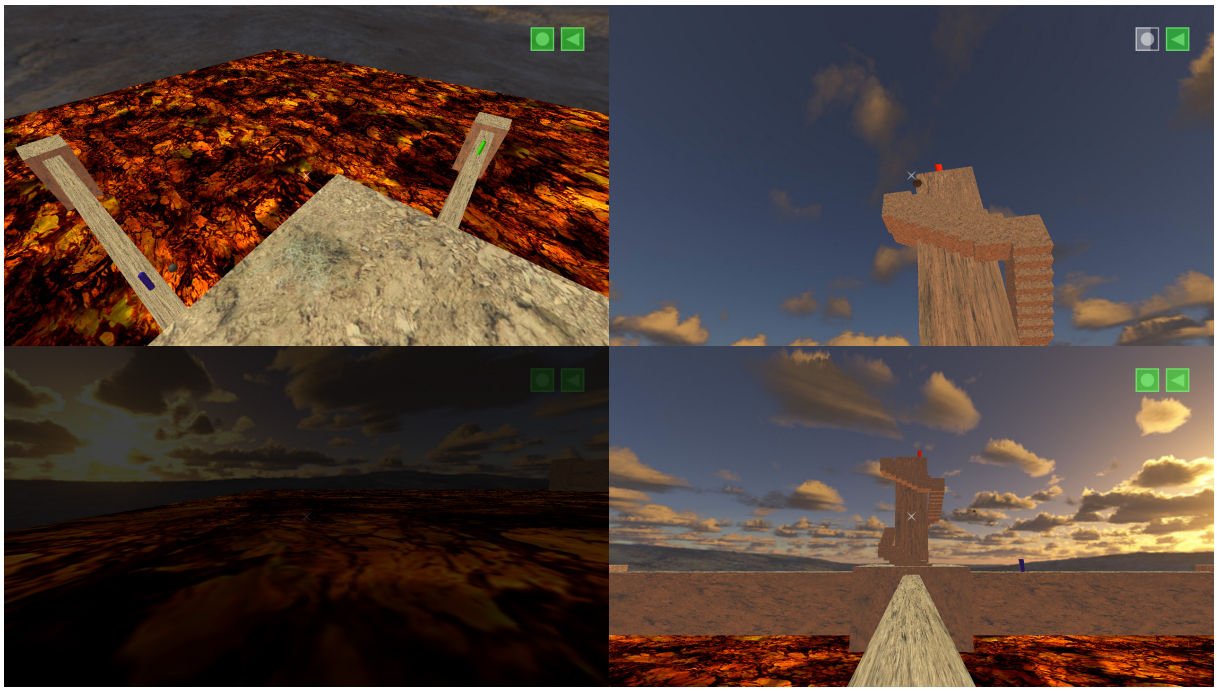
5.1 Quarter One | Dec. 26th - Mar. 20th

5.1.1 Playtest One (Band) | Feb. 21st



- Game engine base
- Core gameplay
- Simple character controller and physics integration
- Basic controller support
- Placeholder content

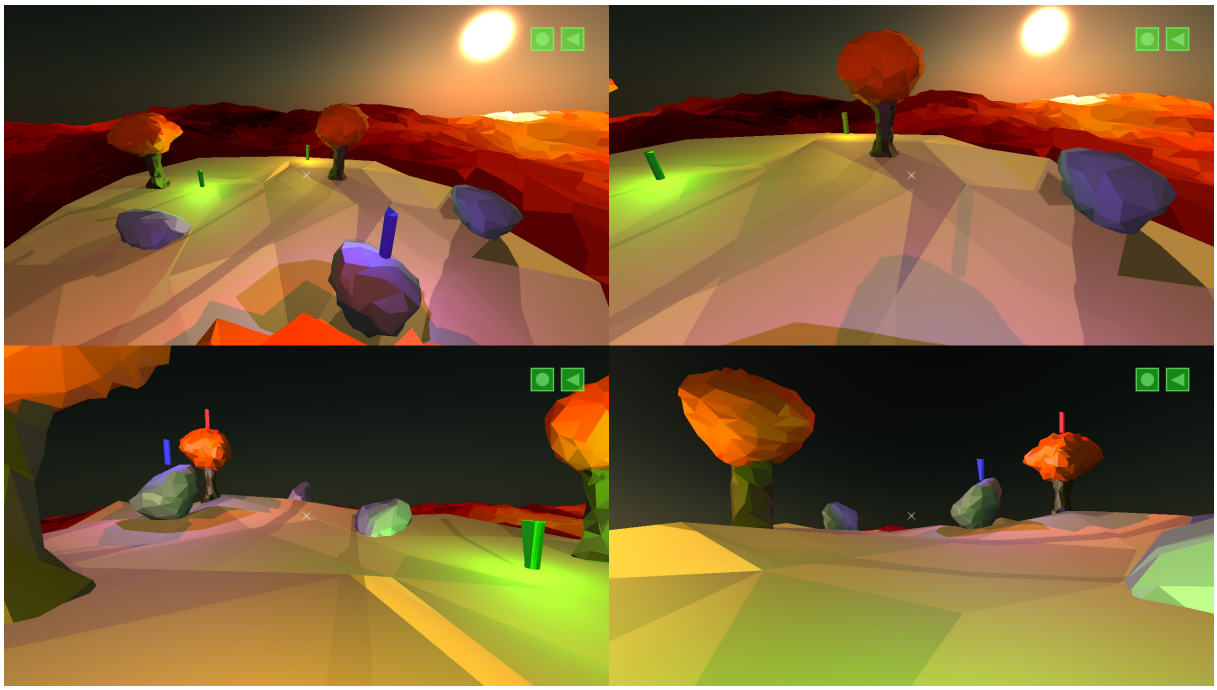
5.1.2 Playtest Two (GDC) | Mar. 4th



- Audio
- Heads-Up Display
- Simple screen-space effects

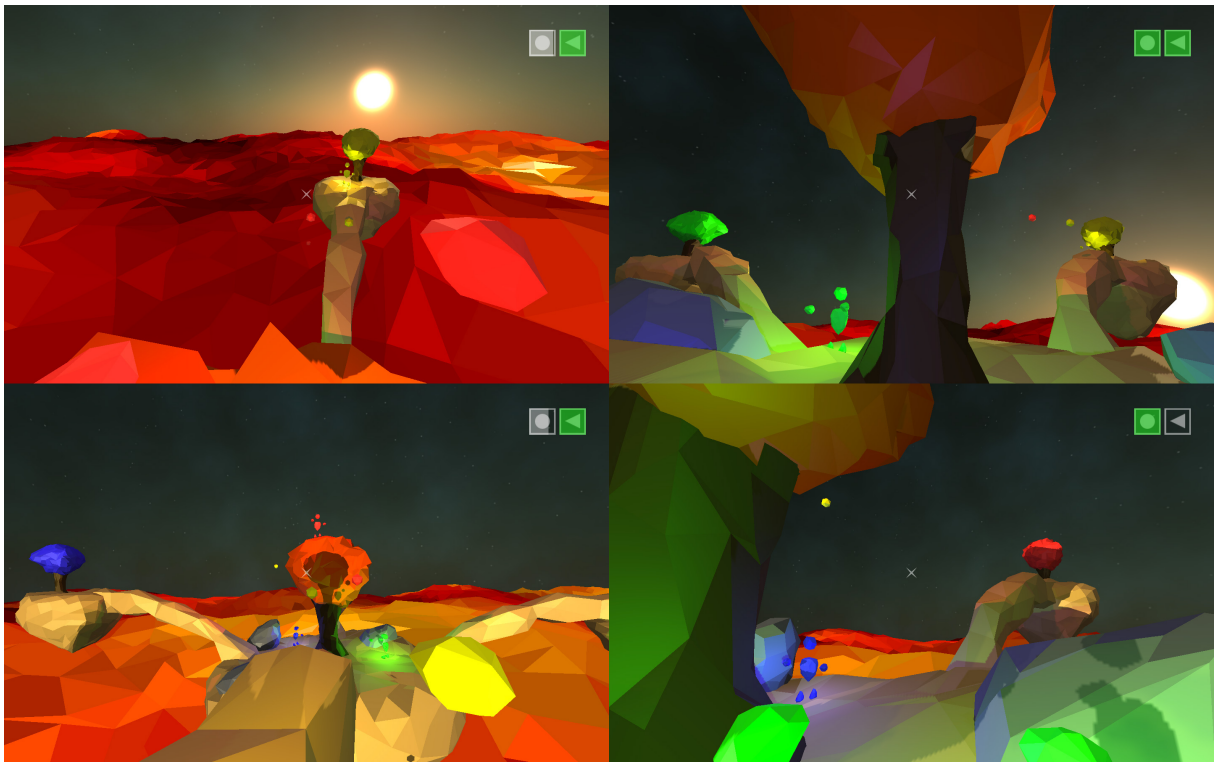
5.2 Quarter Two | Mar. 30th - Jun. 1st

5.2.1 Playtest Three (Band) | Apr. 27th



- Lighting system
- Shadows (including omnidirectional shadows)
- Shader management system
- Updated character controller
- Revised art style
- Sky and lava rendering
- Updated audio
- Level transitions

5.2.2 IndieCade Submission | Jun. 1st



- Menus
- Text rendering
- Scoring
- View frustum culling
- Efficient sky rendering
- Animated characters
- Night sky
- Controller bindings / autodetection
- End / 'win' screen
- Full music

6

Feature Details

6.1 Rendering

6.1.1 Splitscreen

In order to display multiple first-person perspectives in a local multiplayer game, splitscreen rendering is performed. In order to achieve the effect, multiple virtual cameras are used. During each frame, the cameras are iterated over, and a render pass is performed using their view and projection matrices. The area of the window that is rendered to is restricted by setting the OpenGL viewport (performed by calling `glViewport()`).

Splitscreen rendering adds overhead to the rendering cost, though it is not too significant. Due to the viewport restriction, the number of rendered pixels does not increase, and due to view frustum culling, each object in the scene does not necessarily need to be rendered from each perspective.

6.1.2 Shadows

Shadows are implemented using shadow maps [opengl-tutorial.org, 2013]. Prior to rendering objects from each camera's perspective, one or more simplified render passes are performed for each shadowing light. On each pass, all objects within the light's view volume are rendered using a simplified shader into a depth buffer. These depth values are compared on subsequent render passes to determine if a pixel is lit or shadowed for each light source.

View volumes are computed differently for each type of light. For directional light sources (such as the sun), the volume is generated from a view matrix computed from the light's direction and an orthographic projection matrix. For spot light sources (such as those generated with the "push" attack), the volume is generated from a view matrix computed from the light's position and orientation, and a perspective projection determined by the light's cutoff angle. In order to achieve shadowing from spot lights, six render passes must be used. Volumes are generated along the positive and negative axes (with the origin at the position of the light), using a perspective projection with a ninety degree field of view. Instead of rendering into standard depth buffers, each of the six passes is stored into the face of a depth cubemap, in order to gain full coverage of the scene from the light's location.

6.1.3 Sky

The sky is computed in screen-space, using separate processes for the day (atmosphere) and night (stars), and blending in-between the two. For efficiency, the sky is rendered at maximum depth, after all other geometry for each camera, in order to make use of early depth testing [OpenGL Wiki, 2015a] (which is not officially part of the OpenGL specification, but is supported by many GPUs). When supported, early depth testing allows for the fragment shader to not run on fragments that are occluded by other geometry. As the sky shader is relatively expensive, early depth testing allows for a decent speedup.

Day

The atmosphere color and sun are calculated in real-time, using approximations to simulate the scattering of light in the atmosphere. First, a Rayleigh, Mie, and spot factor are computed using a phase function. Second, an "eye depth" value is computed based on the thickness of the atmosphere and the viewing direction of the camera. Third, a number of samples are taken between the eye's location and the edge of the atmosphere, with absorption calculated at each sample and summed up. Finally, the atmosphere color is computed as a function of the total light absorption and the Rayleigh, Mie, and spot factors.

The end result is that the sky changes colors as a function of the sun's location, creating sunrises and sunsets.

The algorithm for the atmospheric calculations is based on an online article. [Boesch, 2011]

Night

The night sky is significantly simpler. A set of pre-generated textures (created in Spacescape [Peterson, 2014]) is sampled as a cubemap. During the day, the night sky is hidden behind the atmosphere. During sunrise and sunset, the night sky is blended with the atmosphere, to create a smooth transition.

6.1.4 Lava

The lava is loosely based on a lava shader that was found online [TheGameMaker, 2007], though it has been heavily modified. The base mesh for the lava was generated in Blender, which is animated using trigonometric functions in the vertex shader. The color and brightness are defined by textures, which move around independently of each other in order to achieve the effect of having the lava "flow".

6.1.5 Text

Text is baked into a font atlas using stb_truetype [Barrett, 2015], and can be rendered in either screen-space or world-space. Screen-space rendering is achieved by rendering individual characters into a temporary texture which is immediately drawn to the screen. World-space rendering is achieved by calculating the space required for the entire string, creating a texture to match, and rendering the characters into the texture. Once the texture has been generated, it is passed off to be stored in an object within the game world.

6.1.6 Debug

Debug information from the physics system can be displayed using a specialized debug renderer. Over the course of the physics simulation of each frame, debug information is collected and stored into vectors. During the late stages of rendering, this information can be displayed by packing it into a buffer and sending it to the GPU to be drawn as lines / points. Because the geometry changes frame to frame, it must all be passed to the GPU each time it is drawn. This is not very efficient, but it allows for debugging of the physics system and can be toggled both at compile-time and run-time.

6.1.7 HUD and Screen-Space Effects

Both a heads-up display and screen-space effects can be rendered in a final pass for each camera. Elements of the heads-up display are implemented as textures, which can be tinted / cropped at run-time to achieve different effects (e.g. an icon 'filling up' or changing color). Their placement is set as a two-dimensional location specified as a percentage across the screen. This allows the elements to be displayed reasonably well at different resolutions and aspect ratios.

Simple screen-space effects can be applied, allowing for clean transitions between levels (achieved by fading the screen to black, changing the level, and fading back). They also allow for general tinting of the output color, which is used to color the screen of the winning player at the end of each round.

6.2 Physics and Animation

6.2.1 Physics

The Ground Is Lava! uses Bullet Physics [Real-Time Physics Simulation, 2015] for its collision detection and physics simulation. Players are represented in the game world as a capsule shape that floats on top of an invisible spring. This allows players to collide with objects in the world, while at the same time being able to smoothly walk across various uneven surfaces. Ghost / trigger volumes are used for both the "push" attack, and to determine when a player has fallen into the lava.

6.2.2 Animation

The stone golem characters in the game are visually represented by separate models for the head, torso, hands, and feet. Each section of the body is attached to the physical representation of the player by invisible springs, allowing the models to respond to changes in momentum. The head follows the orientation of the player's view, causing it to tilt when a player looks up or down. The hands have a slight up/down and forward/back movement as players walk around, achieved by adjusting the resting positions of the springs they are attached to. The feet move in an elliptical pattern based on players' velocities, so that they move in the direction of travel. In order to match the feet to the surfaces within the world, raycasts are performed in order to determine their correct height.

6.3 User Interaction

6.3.1 Menu

Menu Items

The menu exists as world-space text. The mouse is used to select menu items, after which the camera moves to display additional options. It was decided that menus should be implemented this way for a few reasons. First, it makes interacting with the menu more interesting and immersive than it would be to just click on items in screen-space. Additionally, it was relatively easy to implement.

The menu items themselves are just textured quads that exist in the game world. The physics system's raycasting abilities are used in order to determine when the mouse is hovering over menu items. First, the world-space positions of the mouse on the near and far planes are calculated by using `glm::unProject()`. Next, a ray is fired from the near point to the far point, and it is determined if any objects are hit. Finally, if a hit is detected and the object has registered itself as a clickable item, its callback function in run in order to take action.

Camera

The camera moves smoothly around the scene when transitioning between menu items. The effect is achieved by linearly interpolating camera positions and performing spherical linear interpolation on the camera orientations (represented as quaternions), using a time value that is filtered through the `glm::smoothstep()` function.

6.3.2 Controllers

The characters in the game are controlled with joysticks / controllers, which are managed via the GLFW [The GLFW Development Team, 2015] input API. In order to support the large number of controllers / joysticks available to users, a controller binding system was created that abstracts away the specifics of each device, presenting the game logic with a simplified interface that represents the intentions of each player.

For the IndieCade release of the project, controller bindings were implemented for the most common controllers (Xbox 360, Xbox One, PS3, PS4) for each operating system's most common drivers. Next, a system was created for automatically detecting the type of controller connected (by checking controller names / input capabilities) and assigning the correct binding. Finally, a simple interface was designed in the game's menu for manually selecting a binding for each controller in case of automatic detection failure.

6.4 Shaders

6.4.1 Attributes

In order to increase compatibility across shaders, consistent layout qualifiers [OpenGL Wiki, 2015b] are used for attributes. This allows one game object to switch between multiple shaders (e.g. in order to do a shadow map pass) with ease.

6.4.2 Uniforms

While developing the shadowing system for the project, issues were run into with the original uniform management system. Due to the design of the system, it was possible to accidentally overwrite uniform values in a shader by passing the wrong type of data to a `glUniform()` call. In order to prevent such issues and headaches later, it was decided that a new uniform management system should be created.

At shader program link-time, properties of each uniform (including type information) are collected via calls to `glGetActiveUniform()`, which are stored in objects in the engine. This allows run-time type checking of uniforms that are attempted to be set. By caching the uniform data in main memory, the system also improves efficiency by only sending uniform values across to the GPU when necessary.

6.5 Assets

6.5.1 Importing

Assets are imported via asset manager classes, which provide a simple interface to the rest of the engine, allow for caching, and improve robustness. In order to load an asset, one simple function must be called, with the name of the asset provided. Internally, the asset managers deal with the actual I/O operations, error handling (including falling back to default, internally stored assets on failure), and caching.

The asset fallbacks prevent crashes when an asset fails to load, while still making it obvious that there is an issue. Asset caching is performed by holding onto a shared, reference counted pointer of the asset and storing it into a hash map. When an asset is requested, the map is checked to see if the asset was already loaded in order to improve load times. The caches can be cleared if memory restrictions are tight.

6.5.2 Reloading

A side effect of caching assets (along with their source paths) is that they can be reloaded at runtime. The engine currently reloads all assets in debug builds when the game window loses and regains focus. This causes changes made outside of the game to be reflected live. For example, shaders can be modified in a text editor, with their results visible immediately. This allowed for quick shader development and iteration.

6.6 Audio

The Ground Is Lava! uses FMOD [Firelight Technologies, 2015] for audio, which has support for three dimensional sounds and looping tracks. All of the audio was designed to have a "16-bit" style in order to fit with the low-poly art style of the game.

6.6.1 Music

Music was written for the project by Max Linsenbard. A single track exists for the menu, two different tracks are used during gameplay, and one track is played at the conclusion of a game. Fades are supported in order to smoothly transition from one track to another.

6.6.2 Sound Effects

Sound effects were generated with ChipTone [SFB Games, 2015]. The sounds are triggered by events fired by the game engine, which are picked up by audio components associated with different game objects. Sounds can have position and velocity information, which allows for volume changes based on the distance from each camera to the sound source (and can even have a Doppler effect applied when only one camera is present).

6.7 Code Management

6.7.1 Component-Based Game Objects

Objects within the game are represented by a class that acts as a shell. No functionality is implemented in the `GameObject` class itself - it merely exists to hold onto separate components for different systems in the game (such as graphics, physics, game logic, etc.).

Component based game objects have some great benefits - creating an object within the game that mixes certain properties of other objects is as easy as attaching the correct components together (for example, causing an object to emit light is as simple as attaching a `LightComponent` to it). They do have some downsides, however. Namely, communication between components attached to the same object (and even components attached to separate objects) can require run-time dynamic casting, which is not ideal.

6.7.2 Data Ownership

It is crucial to manage data ownership in C++ applications in order to achieve efficiency and prevent memory leaks. The ownership of different data was constantly considered during development. No raw pointers are ever directly allocated, and raw pointers are only used when necessitated by an external library.

Within the game, stack allocations are preferred over dynamic memory allocations, in order to avoid the overhead of allocating memory from the operating system. When dynamic memory is required, it is always managed by some form of smart pointer. `std::unique_ptr` is preferred, as it provides a clear understanding of ownership - whatever holds onto the pointer owns the underlying data. For resources that need to be shared, `std::shared_ptr` is used (e.g. for caching loaded game assets).

Whenever temporary access is needed to a resource, a standard C++ reference is preferred over using a pointer type. This prevents resources from being held onto longer than necessary.

6.7.3 Source Control

Though there was only one developer on the project, git was still extensively used for source control. It helped to enforce code quality (by encouraging small self-code reviews on each commit), provide a timeline of changes, and act as a method of data backup.

6.7.4 Assertions

Assertions are prevalent in the project's codebase. Their use provided clear expectations of function input data and allowed for early detection of game-breaking problems. Inclusion of assertions reduced development time by limiting the amount of time spent debugging and tracking down issues.

6.7.5 Logging

Logging was also used extensively. Using a logging library allowed for different log categories, and provided a means for removing log statements in release builds (allowing for zero performance impact at runtime).

7

Results

The Ground Is Lava! is a fully-functioning video game, playable on Windows, OS X, and Linux. It has been submitted to IndieCade [IndieCade, 2015] (an independent game developers' conference), where it will be judged alongside other independently created games.

7.1 Being Fun

One of the main goals of the project was to create a fun experience. While it may be hard to quantify fun, the responses received during later playtests and gameplay recording sessions (yelling, trash talking, etc.) indicated that players were thoroughly enjoying the experience of playing The Ground Is Lava!

7.2 Reusability

The game was written in a manner that separated specific game logic from the underlying systems used to power the experience. The game engine could be reused and expanded upon to create additional games, with significantly less effort than was required to create The Ground Is Lava!

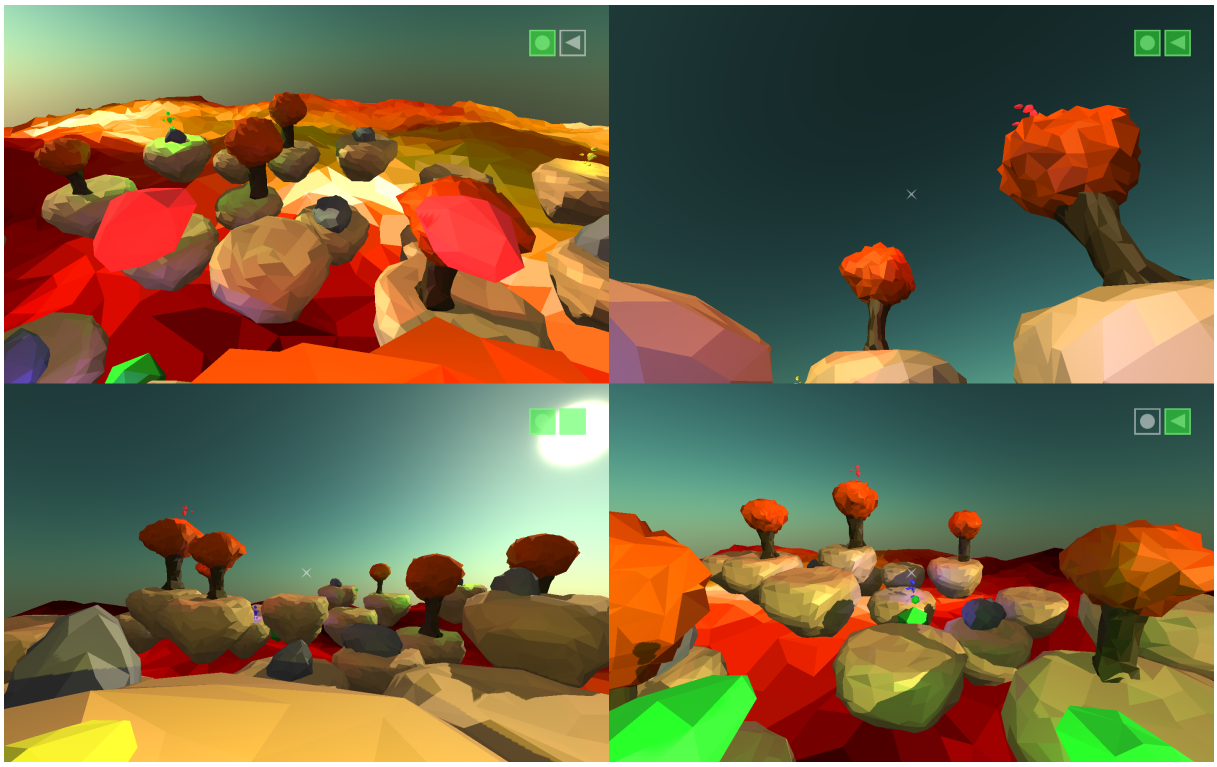
7.3 Art Style

The art style of the game was selected to fit the constraints of the project: no dedicated artists, very little experience using content generation software (such as Blender), and a relatively strict time constraint. In order to avoid appearing bland, bright and colorful scenes were created.

7.4 Compatibility

An early target for the project was to develop a game that could potentially be released as a commercial product. In order to make hitting that target a possibility, compatibility (across different hardware, operating systems, etc.) had to be considered during development. The final product is capable of running on any semi-modern hardware (OpenGL 3.2+, 128MB VRAM) on three operating systems (Windows, OS X, Linux) using any combination of four types of controllers (Xbox 360, Xbox One, PlayStation 3, PlayStation 4).

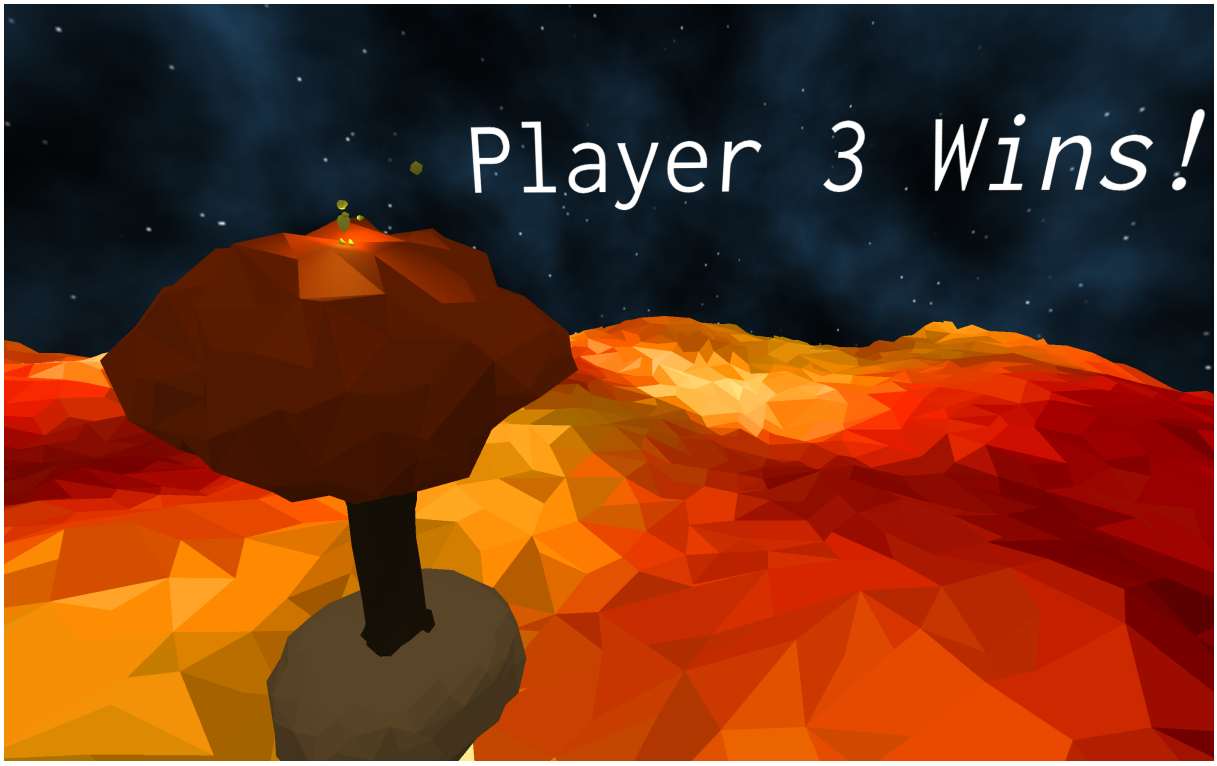
7.5 Screenshots



Real-Time Sky Rendering, Lighting, and Shadowing

The Menu

Player 3 Winning A Round



Player 3 Winning A Game, With Night Sky Visible

8

Conclusion

The development process of The Ground Is Lava! was a wonderful learning experience. It offered insight into the game development process, with lessons on both technical and artistic topics.

On the technical side, much was learned about how to structure, organize, and develop a game engine that is capable of running across different platforms. Time management skills were tested, as deadlines had to be met and features needed to be prioritized.

On the artistic side, the creation of the project mandated the selection and realization of an art style, and the creation of art assets to fill the world. It required the generation of sound effects and composition of music. It forced the design of gameplay, and of the world in which players would interact.

Overall, the creation of The Ground Is Lava! was an incredibly powerful learning experience that allowed for the demonstration of knowledge gained over the course of earning a computer science degree.

9

Future Work

9.1 Gameplay

The Ground Is Lava! was designed to be a relatively simple game in order to account for its short development time. Many new gameplay features could be introduced in order to expand on the existing experience, such as moving platforms, new player abilities, and additional levels.

9.2 Controllers

In order to support any type of controller, in-game binding could be added. Players would be able to attach any controller device, navigate to a menu, and manually specify buttons and axes in order to customize their control setup.

9.3 Efficiency

Dynamic memory allocation is used extensively in the project. Many retail games avoid using the standard heap allocation model, as it is not fast enough. In order to improve efficiency, custom allocators could be used. A pool allocator could be used in the creation of many, same sized structures (such as objects of the `GameObject` class), and a stack allocator could be used for memory needed on a frame-by-frame basis.

9.4 Commercial Release

With some cleanup and polish work, the game could be released commercially on a digital distribution platform. Extensive testing would need to be done in order to ensure that players would have a consistent experience.

References

[Assimp Development Team, 2014] Assimp Development Team (2014). *Open Asset Import Library*. http://assimp.sourceforge.net/.

[Barrett, 2015] Barrett, S. (2015). *stb*. https://github.com/nothings/stb.

[Boesch, 2011] Boesch, F. (2011). *Advanced WebGL - Part 2: Sky Rendering*. http://codeflow.org/entries/2011/apr/13/advanced-webgl-part-2-sky-rendering/.

[Boneloaf, 2014] Boneloaf (2014). *Gang Beasts*. http://gangbeasts.com/.

[Firelight Technologies, 2015] Firelight Technologies (2015). *FMOD*. http://www.fmod.org/.

[G-Truc Creation, 2015] G-Truc Creation (2015). *OpenGL Mathematics*. http://glm.g-truc.net/.

[IndieCade, 2015] IndieCade (2015). *IndieCade*. http://indiecade.com/.

[Matt Makes Games, 2014] Matt Makes Games (2014). *TowerFall Ascension*. http://www.towerfall-game.com/.

[Messhof, 2014] Messhof (2014). *Nidhogg*. http://www.nidhogggame.com/.

[opengl-tutorial.org, 2013] opengl-tutorial.org (2013). *Shadow Mapping*. http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/.

[OpenGL Wiki, 2015a] OpenGL Wiki (2015a). *Early Fragment Test*. https://www.opengl.org/wiki/Early_Fragment_Test.

[OpenGL Wiki, 2015b] OpenGL Wiki (2015b). *Layout Qualifier*. https://www.opengl.org/wiki/Layout_Qualifier_%28GLSL%29#Vertex_shader_attribute_index.

[Peterson, 2014] Peterson, A. (2014). *Spacescape*. http://alexcpeterson.com/spacescape/.

[Real-Time Physics Simulation, 2015] Real-Time Physics Simulation (2015). *Bullet Physics*. http://bulletphysics.org/.

[SFB Games, 2015] SFB Games (2015). *ChipTone*. http://sfbgames.com/chiptone/.

[Smithsonian Institution, 2012] Smithsonian Institution (2012). *The Art of Video Games*. http://www.americanart.si.edu/exhibitions/archive/2012/games/.

[TheGameMaker, 2007] TheGameMaker (2007). *Lava Shader*. http://threejs.org/examples/webgl_shader_lava.html.

[The GLFW Development Team, 2015] The GLFW Development Team (2015). *GLFW*. http://www.glfw.org/.