ASCENT

- A six-degrees-of-freedom first-person shooter game -

Project Report

TDA335 - Programming project

Chalmers University

Contents

Preface

This document presents to you the report written for a programming project required for a course of the same name at Chalmers University of Technology in Gothenburg, Sweden. It contains detailed explanation of the design of the project, as well as its implementation.

Due to personal preferences, I chose the work for this course to consist of building a game prototype and implementing the entire base engine from scratch. I felt this encompassed a very broad range of the knowledge I accumulated during my time at Chalmers. It was also meant to serve as a drive for my personal benefit and inspire me to take a project of this magnitude from start to finish. In this, it was a success.

The game prototype is heavily inspired by the classic game Descent, hence its name. It uses similar game play mechanics and extends them with modern features like procedural generation.

Throughout this document you will read about the technologies used during development in chapter 1, the design choices made for everything from enemy ship generation to player input in chapter 2, explanations for the engine and game layer implementation in chapter 3, and a short list of simple additions to be made in the future in chapter 4. Finally, we summarize the work in chapter 5.

I would like to thank my supervisor, Staffan Björk, for his guidance during this project. I would also like to express my gratitude towards the faculty at Chalmers for providing me with the skill set required to finish a project like this. Last but not least, I would like to thank you, for taking the time to read this report. I hope you enjoy it.

Flavius Alecu

December 4, 2016

Chapter 1

Introduction

Video games, being a popular entertainment medium, continue to make advancements in visual fidelity as well as design and technical achievements. Developing a game from the ground up, including all technologies required for the final product to run, often on multiple platforms, requires a vast amount of time and resources. This project aims to design and implement a game prototype inspired by the 1995 game Descent [5] and the roguelike genre [19] with as few off-the-shelf resources as possible. This is accomplished mainly through procedural content generation (PCG) [13] and the choice of a simple design.

All source code was developed for this project with a few isolated exceptions. Bullet [4] physics engine was used to simulate collision detection, the SoLoud [10] audio engine provided the needed support to play back sound effects and the background music, and a previously written utility library was embedded for various math and linear algebra operations.

The background music was composed specifically for this project and audio data from Freesound [6] was utilized for the various sound effects. All art is procedurally generated at runtime, as explained in more detail in chapter 3.

Microsoft Visual Studio 2012 and the Microsoft Visual C++ compiler [3] was used as the development environment and the source code was stored in a Git [7] repository at Bitbucket [1]. The engine developed uses Simple DirectMedia Layer (SDL) 2.0 [2] as the core library and is based on the OpenGL 4.3 [9] graphics API, for which the requirement is justified in detail in chapter 3.

Chapter 2

Design

2.1 Procedural Content Generation

Procedural content generation is a process during which assets are generated with the help of pseudo-random algorithms. The result is never truly random due to the nature of computer processors, but a large enough range of variation can be achieved to make the result be perceived as random.

For this project, a set of resources are created programmatically when the game starts up. The variables, and ranges for these variables, used in this process are then modified by seemingly arbitrary series of numbers generated from a seed value. The seed itself is based off the internal clock value of the computer at the point in time when the game is launched. This method provides a wide enough range of results to avoid noticeable repetition. It also ensures that every time the game runs the assets generated look slightly different from previous runs.

The following sections will provide more details on how PCG is used to allow for the creation of various types of content used in Ascent.

2.1.1 Level Design

The level design in Ascent was deliberately kept simple in order to not divert focus away from the main ambition of the project, namely the technical implementation. Each level is merely a set of rooms connected by tunnels, as can be seen in figure 2.1.

The rooms are randomly sized rectangular cuboids and the tunnels follow the shortest path between the openings in each room. However, to avoid straight paths, the tunnels are axis aligned. The player always moves along one, and only one, of the three spatial axes and never diagonally when moving through a tunnel. This introduces hard 90° corners and edgy stair-like passages, depicted in figure 2.2. Moreover, the side of a room where the opening connecting it to a tunnel is placed

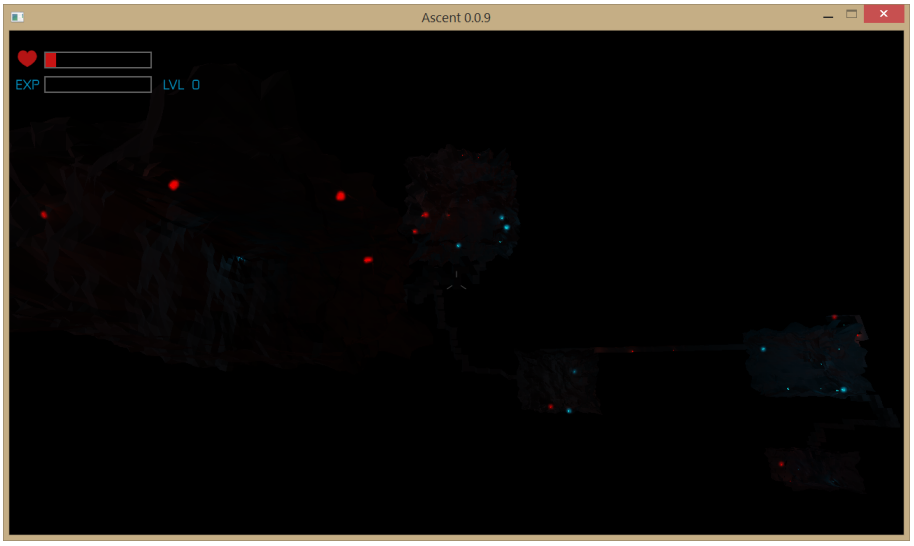

Figure 2.1: A debug render of a randomly generated level.

is chosen randomly, forcing the tunnel to follow the outside of the room in cases where the adjacent room is on the opposite side of the opening.

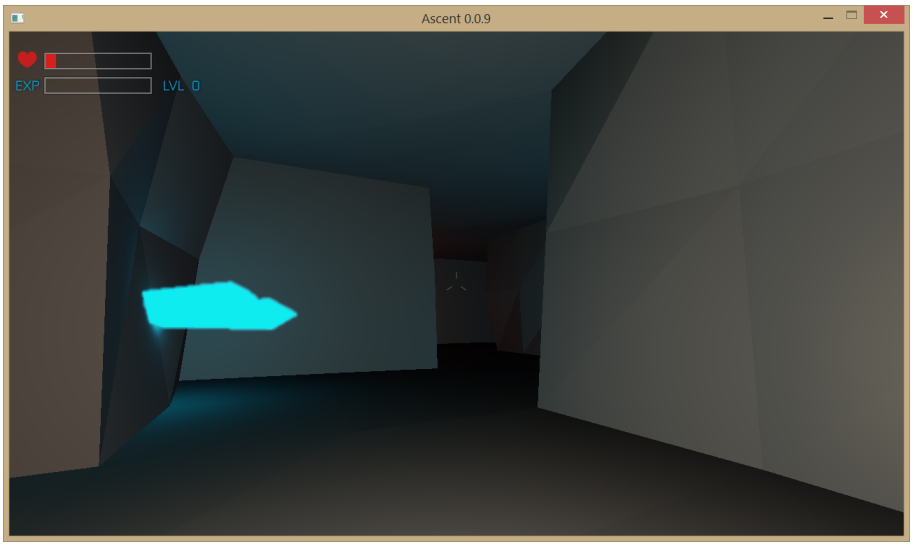

Figure 2.2: Harsh 90° corners add variety to the tunnels connecting the rooms.

2.1.2 Enemy Ships

In broad terms of visual appearance there is only one type of enemy ship. The ship is however composed of a set of randomized segments that result in a wide

array of different looking enemies. While the behavior of the ships can differ, as explained in later sections, the randomly generated models are purely cosmetic.

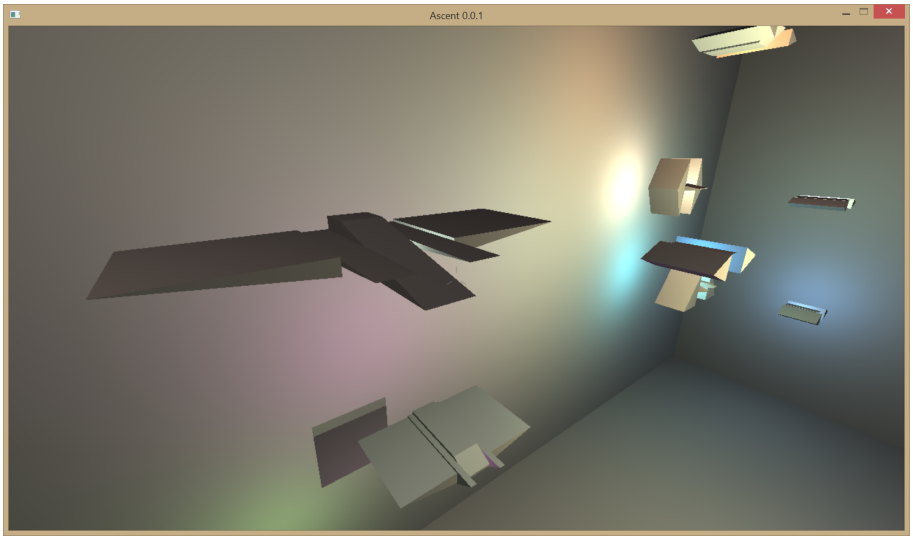

Figure 2.3: An example of different ships the procedural geometry generation algorithm can create.

Each ship is comprised of three sections: a center rectangular cuboid, two mirrored rectangular cuboids (one on each side of the center piece), and two mirrored triangular prisms serving as the outer wings. The sizes for the various shapes are randomized within predefined ranges. This has proved enough to generate the variation required for this project.

2.1.3 Enemy Weapons

In Ascent all bullets are created equal. There is only one type of weapon for all enemies and every bullet does the same amount of damage. There are however different ways in which an enemy ship can attack the player. This is dictated by three variables: the number of bullets fired with each shot, the cool down time between each shot, and a sequence pattern.

The simplest version of the enemy attack variation is a single bullet fired with each shot towards the player. A second, less sensitive to accuracy, attack fires five bullets simultaneously. One bullet travels straight towards the player while the remaining four move in a conical pattern that grows larger as the distance from the firing ship increases. Lastly, we have a bullet pattern almost identical to the previous one only with a narrower cone for the outer four bullets and no center bullet at all. These weapon modes are demonstrated in figure 2.4.

Ships fire their bullets in a random sequence where each sequence consists of one to ten shots. The firing mode is randomly chosen for each ship at level start and stays constant until the ships are destroyed.

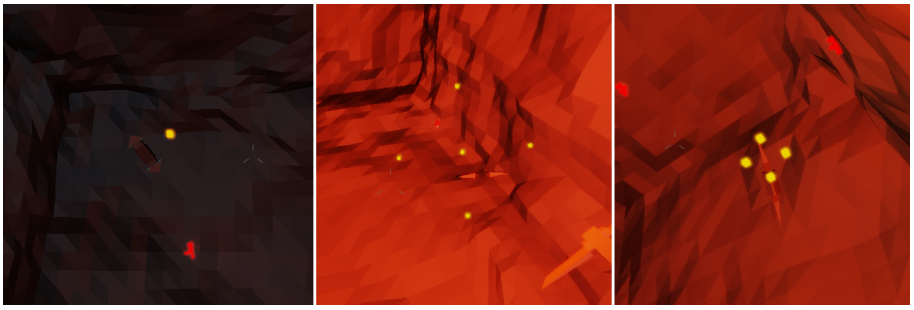

Figure 2.4: Low tier enemy weapon firing one bullet at a time (left), middle tier weapon firing five bullets in an expanding cone (middle), and top tier weapon firing four bullets in a narrow conical pattern towards the player (right).

In addition to the previously mentioned parameters, the time between bullets being fired, their velocity, and their accuracy can also vary. These settings are based on the intelligence level of the parent ship and will be explained in more detail in the following section.

2.2 Artificial Intelligence

The AI system in Ascent is based on five difficulty levels randomly assigned to enemy ships when they spawn. Each level adds a minor set of improvements to the behavior of the ship. These improvements change the movement speed, shooting behavior, and response time.

Ships assigned difficulty level 1 or 2 will be static and not move from their initial spawn location. The delay between each shot is four and three seconds for level 1 and 2 respectively. Additionally, a relatively high modifier is used to perturb the aiming vector, which results in poor accuracy. The bullet velocity is also lower than that of higher difficulty level ships. Overall, this makes for reasonably easy targets.

Enemy ships stay dormant until the player enters their line of sight. While visibility is omnidirectional, enemy ships cannot perceive the player through walls. Thus enemies begin to act as soon as the player approaches a room from one of its connecting tunnels. Once an enemy sees the player, it needs to rotate its ship into alignment. Only then can it fire the weapon. The speed with which ships rotate to align themselves before firing is the response time affected by the ship difficulty level.

Difficulty levels 3 through 5 have ships moving randomly around the room at progressively faster speeds. The other parameters are similarly modified in order to improve accuracy, raise bullet and ship velocity, increase firing rate, and lower response times. In essence, with each difficulty level enemies get increasingly stronger and more difficult to survive once encountered.

2.3 Progression

Each level consists of three to six rooms. At the start, the player spawns in the first room and can reach the rest of the level through various tunnels. Once the player has found and defeated all enemies, the level ends and a new one is randomly generated.

Each room, except the one where the player spawns, has a random number of one to ten enemies. A pseudo-random formula based on the player's experience level is used to determine how many ships to spawn. The more experience the player has, the higher the number of enemies spawned in each room.

Throughout the levels in Ascent the player can find two types of crystals that can be gathered. Red crystals regenerate the player's health and blue crystals contribute to the experience level. The red crystals have no effect when the health meter is full. However, the player is encouraged to immediately start gathering crystals by spawning with a very low health meter at game start. In contrast, the blue crystals are always valuable as the player level is incremented every time the blue meter is filled.

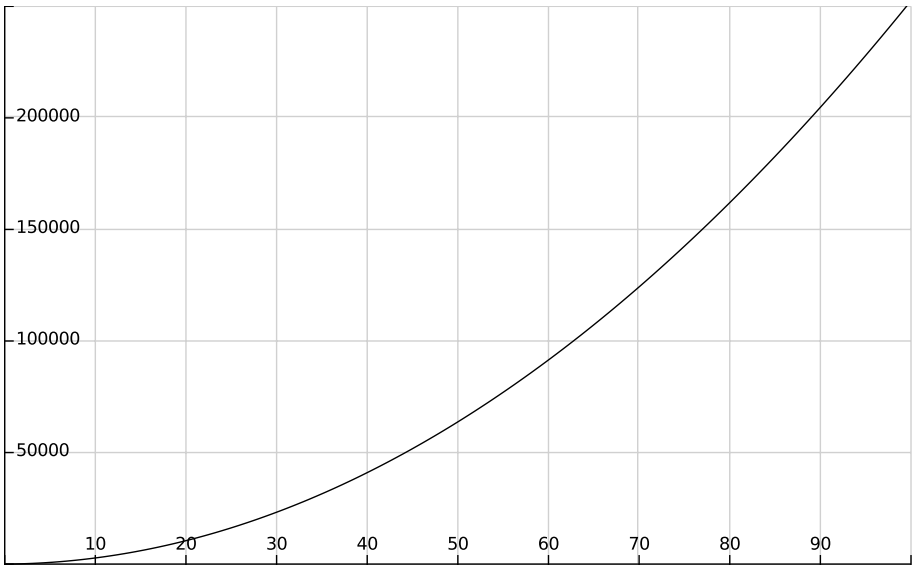

Figure 2.5: The formula for leveling up in Ascent is open ended with no theoretical limit for a maximum level. The x-axis shows the level and the y-axis shows the amount of experience required.

With each new level, the player weapon cool down time is decreased which raises the fire rate. The formula used to level up was purposefully developed to allow fast advancements in the beginning while slowing down progression at higher levels without ever hitting an artificial maximum level. This is meant to keep the game open ended, although in practice the progress slows down to the point where a hard limit is perceived nonetheless.

The weapon level formula is $25 * level * (level + 1)$ and can be seen plotted in figure 2.5. It is clearly evident that advancing to a new level becomes slower the higher level a player is. Moreover, the decrease in weapon cooldown responsible for the increase in fire rate is governed by the formula $1 + 1/ - (level/2 + 1)$, plotted in figure 2.6. This was specifically designed to lack a hardcoded upper limit yet ensure diminishing returns. A very low weapon cool down time at higher levels results in a continuous stream of bullets where an improvement has a barely noticeable effect compared to lower levels.

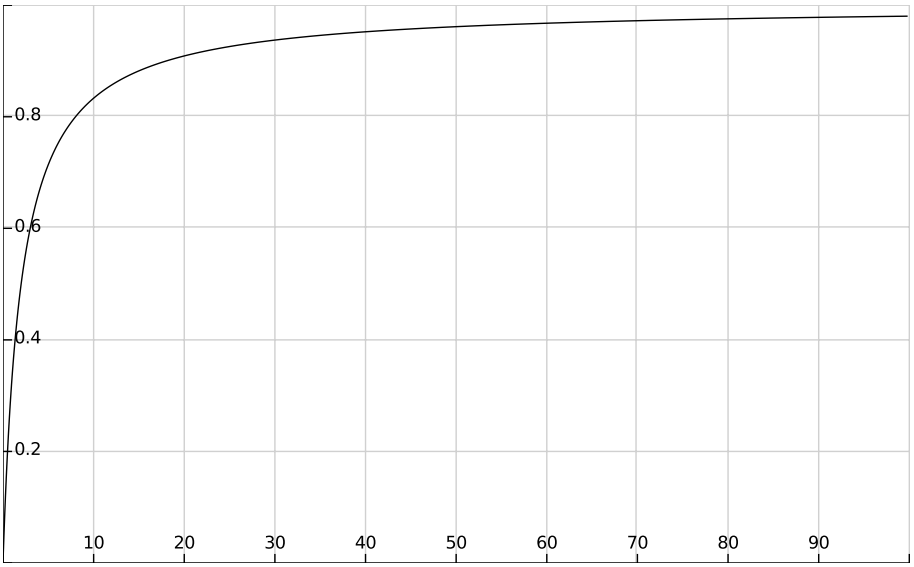

Figure 2.6: This formula is used to interpolate the player weapon's cooldown time between an initial high value and a minimum, unreachable value as the player's level increases.

2.4 Input

One of the strongest similarities between this project and Descent is the six degrees of freedom the player has when it comes to movement. This, together with the first-person view, serves to immerse the player in the outer space setting of both games.

Camera roll is a relatively uncommon feature in most games outside of those overlapping the flight simulation genre. It is even more unlikely to be found in games whose level design resembles that of Ascent with its narrow and claustrophobic spaces. While this feature was included because it follows the true spirit of the Descent games, it contributes with an additional, albeit subtle, benefit. The level design, which unfortunately suffers from slight monotony due to the implementation choices justified earlier in this chapter, receives an elusive boost in perceived complexity. This happens as a consequence of the camera roll mechanic

not being common in popular games, as well as the six degrees of freedom movement simulation being a relatively foreign experience to most players. The lack of a static orientation, which is ubiquitous in the majority of games released today, makes it hard to recognize previously seen areas. The result is a simple level layout that can easily confuse players and cause them to get lost if they use the level geometry alone to track their location.

Beyond roll, the player controls offer inputs for ascend and descend as well as the more familiar pitch and yaw changes. Movement loosely simulates a floating spaceship with a slight acceleration and drag. The variables were chosen and tweaked to provide a responsive control scheme and are not physically based.

In order to mitigate the confusion arising from the lack of a predefined camera orientation, the system normalizes itself to align the player's local up vector with one of the world axes as soon as the player directly or indirectly rolls the camera in world space. However, the correction only happens when there is no player input that conflicts with the correction required for the alignment. Continuously aligning the player to the world axes facilitates movement through the narrow corridors as it removes the need for the player to manually make small adjustments every time the ship drifts diagonally.

Chapter 3

Implementation

3.1 Engine Layer

Ascent was designed and written from the ground up with minimal off-the-shelf code. The project uses a unified and tightly coupled layout although the logic is split into a lower level engine and a higher level game layer. This organization emerged both naturally due to convenience as well as through the effort of designing a structure to keep the different code modules organized, easy to work with, and allow for future reuse.

This section delves into the various subsystems of the engine layer and provides more detail on how each one works individually as well as how the interconnection between them allow for the final game to function.

3.1.1 Component System

Ascent was programmed in a iterative manner with no traditional design document setting in stone the specific features required. Naturally this caused several prototypes for various systems to be discarded or refactored as it became obvious that solutions to implementation problems would be difficult to develop. One perfect such example is the amalgamation of different systems required to bring together a working prototype of dynamic in-game objects. The ability to move, collide, and emit light are examples of different abilities required for these objects to possess. It became increasingly apparent that separation of logic was needed in order to keep the project tidy and maintainable.

Component frameworks are a common and popular design methodology used in game development, as can be seen in, for example, the successful Unity engine [12]. It allows for logic to be decoupled from objects and kept in isolated modules called components. This has several advantages. Each component is standalone which makes it easier to maintain and extend. Performance can be gained due to

the data oriented approach of processing multiple components of the same type without the added overhead of code irrelevant to the components internal logic. It also allows for dynamic game objects to be treated equally and be distinguished only through what components they posses. For example, the object oriented concept of inheritance, which is known to incur additional performance cost due to unnecessary abstraction and, more specifically, how virtual functions are implemented in C++, need not be used. An enemy object, which in a traditional object oriented implementation would inherit from a base object class, is a simple structure that contains a set of components. The list of components is what defines the structure as an enemy ship, a bullet, a collectible, etc.

For this project a custom component framework was implemented. A simple template interface was created for easy creation, deletion, and querying of an object's components. An example of how this system is used is shown in listing 1.

```
1  if (!ms_player->GetComponent<ColliderComponent>())
2  {
3          ControllerComponent* pController =
4          ms_player->AddComponent<ControllerComponent>();
5
6          if (pController)
7          {
8                  pController->SetAccel(600.f);
9                  pController->SetMaxSpeed(1.8f);
10         }
11 }
```

Listing 1: Example usage of the component system.

The object class contains only the data required by all objects: the matrix storing the position and orientation of the object, the list of components attached to the object, as well as some game play related data like what room the object is in. Arguably, the game play data would be better suited in its own component. While this didn't turn out to be necessary for this project, it would be a trivial extension to make given the current framework. Extending an object into a specific archetype is just a matter of adding the desired components.

The AI component is part of every enemy ship object. It is responsible for moving the ships, firing the weapons, and keeping track of the target: the player. A ship's assigned difficulty level (out of the five mentioned previously in chapter chapter 2) is the main driving force of this component.

The controller component contains the logic required to translate player input into in-game actions. It uses the input system to receive keyboard, mouse and

game pad events, and determines how to use the physics engine to move the parent object as well as fire its weapon. This component also directly modifies the camera position and orientation. It is worth mentioning that the camera can be attached to any object, not just those with controller components. By default the camera is attached to the player object which also happens to be the only object with a controller component, yet the flexibility is there if in the future it would be required for the camera to follow a different dynamic entity, for example during an in-game cut-scene.

While the enemy weapons are part of the AI component, the logic for the player weapon is more complex, and as such, is isolated within its own component. This weapon component is responsible for spawning the bullet objects when the player fires as well as calculate the current level of the weapon based on how much experience the player has gained through the collection of blue crystals.

To stay true to the dark, underground look targeted for this project, there is minimal ambient light in the levels of Ascent. Because of this, bullets, crystals, and even enemy ships subtly emit different colored light in order to be distinguishable from the background. These objects all have a light component attached that contains simple parameters (color, radius, and cutoff) required by the graphics system to render a light.

In addition to lights, every dynamic object contains a collider component. This component is mainly a wrapper around the rigid body object created by the physics engine and is responsible to react to collision events appropriately. The physics system correctly resolves collisions between its internal rigid bodies but this does not directly transfer to the visual representation in our engine. In the collider component we respond to collision events and use the data from the physics engine to adjust the visual representation of those objects. The component also exposes an API for other systems to register an "on collision" callback function. This function is called when a collision happens involving the object's rigid body in order to propagate the event to higher level code. In addition to the above, an optional health calculation is also performed for each collision. Not every collider component will provide its parent object with a health attribute and not every collision will do damage to the object, however. For example, a ship colliding with a wall will take no damage and a bullet will be destroyed on impact, thus no health tracking is needed.

Lastly, any object that requires to be visible in one way or another needs a geometry component. This component encapsulates the vertex, index, and color data defining the geometry of the object. It also wraps the data required by the OpenGL API for rendering the geometry (e.g. handles to the vertex and index buffers).

3.1.2 Shader System

The shader system in Ascent is responsible not only for loading and compiling the GLSL (OpenGL's shading language) shaders but also for creating an accessible interface to the shader variables exposed to the engine (the uniforms in OpengGL terms) as well as validating that the correct type of geometry is used with each shader.

When the game boots all shader coder is loaded and compiled into vertex, pixel, and compute shaders. Combinations of vertex and pixels shaders are then linked into various programs. The vertex shaders are often reused across several programs, since that stage of the shader pipeline is identical in a multitude of situations. Each shader program is given a name which is later utilized when the shader needs to be activated for a certain draw call.

```
1 Shader* pShader = Shader::GetProgram("basic");
2 if (pShader)
3 {
4         pShader->Bind();
5         // render geometry
6         pShader->Unbind();
7 }
```

Listing 2: Example of how a shader can easily be queried by name and used when rendering.

Once a shader program has been linked, the OpenGL API is used to query all available shader variables exposed to the engine. These are stored in hash maps together with their type, size (in case of variable buffers), and the binding required to update the variable in the shader. The shader system exposes a typed interface that can be used to update these variables, as can be seen in listing 3.

```
1 Mat4 mvp = Renderer::GetViewProjectionMatrix() * mat;
2 Shader::SetMat4("g_mvp", mvp);
```

Listing 3: Example of how the model-view-projection matrix can be set on a shader by name.

It is worth noting that while shaders and shader variables are queried and referenced by name, we never really store the full string in our data structures. To avoid wasting space unnecessarily, a 32 bit Murmur hash [17] is created from each string and used instead of the variable length character array.

The vertex shader, being the first programmable stage in the graphics pipeline, is what receives the geometry data from the engine. This data consists of per vertex attributes such as position, normals, and colors. It is paramount that both

the engine and the shader side agree on what attributes are required in order to render the geometry correctly. This is where our shader system validation comes into play.

Once a shader program has been linked, each supported attribute is queried and registered if present in the shader. The result is stored in a bit mask on each shader program object. When a piece of geometry is created, a similar bit mask is calculated based on what data the geometry owns and expects to render. At render time the two bit masks are checked and if not identical an error is given. This system quickly identifies mismatches between geometry and shaders and has proven invaluable during development.

3.1.3 Dynamic Lighting

The Ascent rendering engine supports a large number of dynamic lights being added and removed at any time during the frame. Due to the simplistic nature of the project, we only support point lights, simply because there was no need for any other type of light. Most dynamic objects in the game world have a light attached to themselves to stand out in the scene. This includes all ships, crystals, and every single bullet.

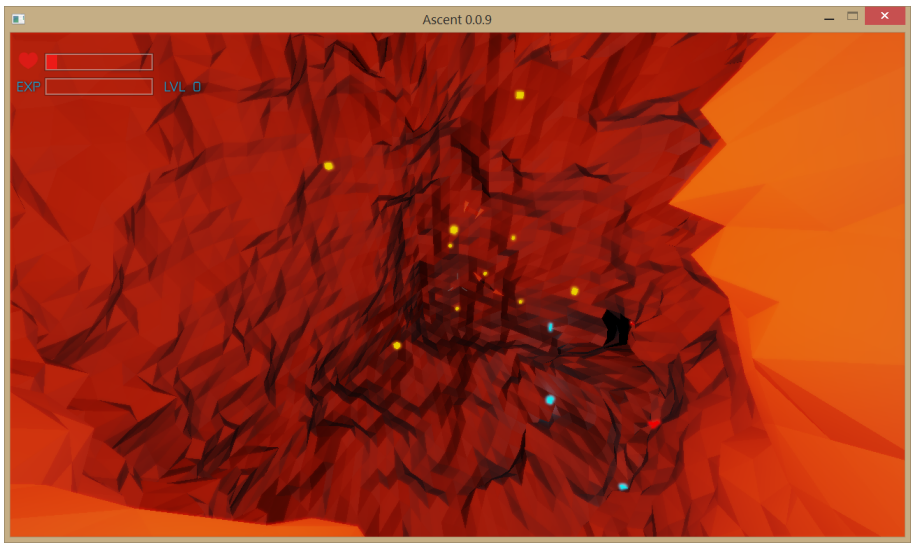

Figure 3.1: A screen capture of the scene being illuminated by multiple lights, demonstrating the dynamic lighting engine.

Each light entry consists of a color, a radius, and a cutoff. Every object, through its light component, is responsible for registering the light data with the rendering engine every frame. This simplifies lifetime tracking for the light entities. When an object is destroyed, it simply stops registering its light and the light disappears from the scene.

At the end of each frame, before we start rendering, the list of all registered lights are passed to the currently bound pixel shader as a uniform buffer. The shader then iterates through the list and accumulates all valid lights into the final color of the pixel. The Phong shading model [18] is used to reach the final color of each pixel.

3.1.4 Visual Effects

It was decided from the beginning that, as a technical challenge, the particle system would be implemented as a compute shader. Since compute shaders were first introduced in OpenGL 4.3 [8], this is the main reason why Ascent requires what appears to be a a graphics API version too modern for its feature set.

In a superficial project such as ours, a particle system implemented on the CPU would easily have been feasible. There are advantages of a compute shader approach beyond the opportunity to gain knowledge, however. The main reason one may want a compute shader implementation is the obvious shift of processing time from the CPU to the GPU. Given the small scale of the project, together with the requirement for a modern api such as OpenGL 4.3, which implies a requirement of a modern and relatively high performance GPU, it would be reasonable to expect that the GPU would be highly underutilized. Shifting processing time from the CPU to the GPU would free up the CPU for additional work while ensuring the likely powerful GPU would have no troubles rendering the required geometry.

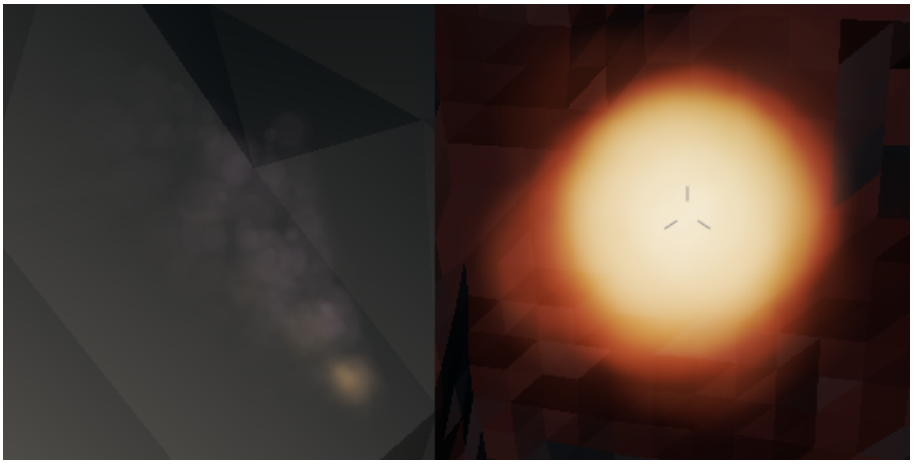

Figure 3.2: Bullet impacts emit dust-like particles (left) and destroyed ships result in a modest explosion (right).

The claustrophobic nature of the level design in this project means that there is a reasonable chance a particle system will trigger close to some other geometry. Bullet impacts spawn a small dust-like effect close to walls and when ships get destroyed a large explosion takes place, as can be seen in figure 3.2. As soon

as these particle systems were implemented they immediately exposed the ugly artifacts caused by the flat particle polygons intersecting with the level geometry and how this kills the volumetric perception of particle effects. To fix this, soft particles [11] were implemented. With soft particles, the scene depth buffer is used to detect the polygon intersection and fade out particle polygons as they get closer to adjacent geometry. The effect is a much improved version of particle effects and can be seen in figure 3.3.

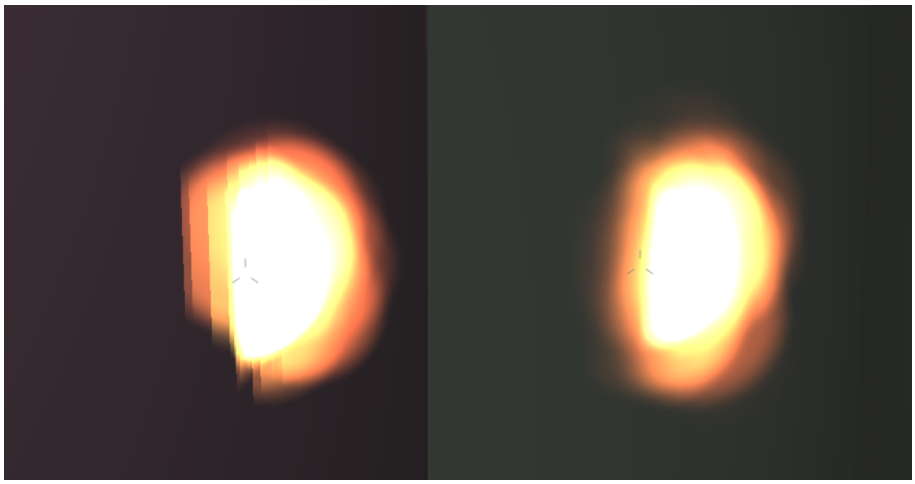

Figure 3.3: A comparison between rendering standard particles (left) versus soft particles (right).

Beyond particle systems, a fake High Dynamic Range (HDR) effect was also implemented as a post process pass in order to increase the visual fidelity of the prototype. HDR imaging is a popular paradigm with recent developments in visual displays and their ability to render high contrast images. Nevertheless, a similar effect called bloom can be reproduced and rendered adequately on traditional displays. We call this a fake implementation because it merely results in a perceived higher dynamic range without introducing additional data outside of the standard definition range.

Bloom is achieved by rendering bright pixels into a separate render target, isolated from the rest of the scene, and blurring this in order for the colors to leak to adjacent pixels. This render target is then composited onto the main scene buffer. The color leak gives the impression of very bright pixels without the need of special displays.

3.1.5 Audio

For audio we utilize the flexible SoLoud engine that loads, plays, and mixes all audio for us. In general, the library takes responsibility for everything required to play a sound through a convenient black box approach.

At the game play level a C++ enum is defined with entries for every single sound required in the game, including the background music. When the game launches this enum is traversed and all sound files are loaded. Playing a sound is then simply a matter of using the enum.

```
1 ms_sounds[BGM].load("audio/bgm.ogg");
2 ms_sounds[BGM].setLooping(true);
3 ...
4 Audio::Play(Audio::BGM);
```

Listing 4: The audio system loads the data internally and client code can easily play a sound by passing the relevant enum identifier.

3.1.6 Physics Engine

Implementation of physics simulation in a 3-dimensional game engine requires significant development time. To avoid diverting attention from the main focus of the project, an off-the-shelf physics engine was chosen for Ascent. Bullet is a well established open source physics engine previously used in successful games as well as visual effects simulations for movie renders [15]. In our project Bullet holds a representation of the game world containing data for all geometry, dynamic as well as static. Under different circumstances the physics data would be a low poly representation of the visual geometry. In our situation, however, all geometry has a low enough tessellation factor that using the same vertex data for the physical representation doesn't incur a high enough cost to be problematic.

For static geometry, such as the level, we create the Bullet data structures required to propagate data to and from the physics engine while manually keeping track of their life span. For all dynamic objects we use the previously detailed component framework. The collider component encapsulates all data required to synchronize the physical representation in Bullet with the game layer logic and the graphics data required to render the object correctly.

3.1.7 Debug Utilities

Several systems were implemented as facilitators for development and don't contribute to the final build of the game prototype. Naturally it is speculation to assume that these systems were valuable because of the time saved during development, especially since that time cannot be measured. Yet by gauging how often they were used we believe they provided a significant aid.

The first thing that was developed, before any of the main game and even engine features, was a debug console. A debug console is a user interface widget that

is hidden by default and can be enabled through a keyboard shortcut. The widget has the ability to show text, for example a debug log, and allows the developer to input commands. The input is then interpreted by the engine and appropriate logic is run for the desired result.

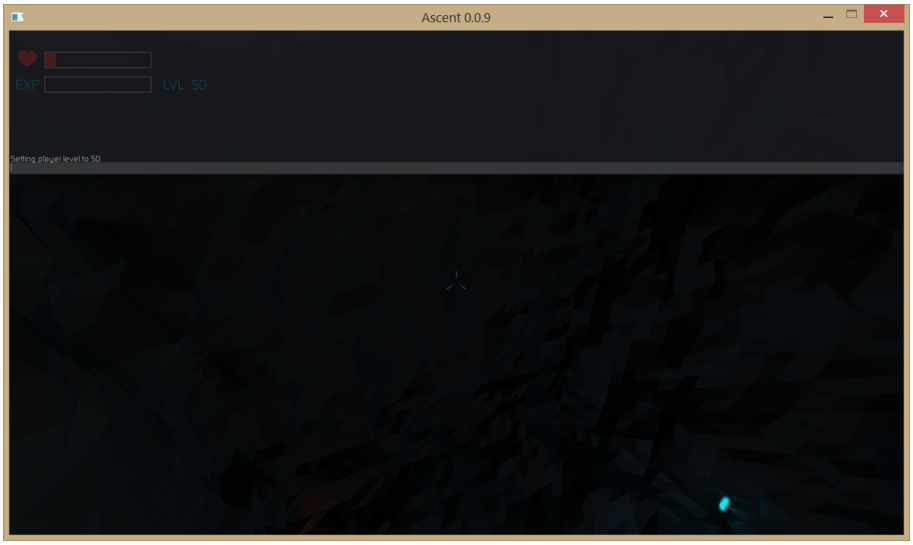

Figure 3.4: The debug console can be used to modify the player's level and other game and engine parameters while the game is running.

The debug console is of little use by itself. This lead to a complementary system being implemented that allows the developer to define dynamic console variables in code which are then exposed at run-time through the debug console. Each variable is represented by a name and a 32 bit unsigned integer. It is trivial to extend the data structure to support additional types, however, the need never arose for this project. Integers, and consequently booleans, proved to be enough for our use cases.

A console variable can easily be added anywhere in the code base through one of two macros. The first macro takes the variable name as the single parameter and initializes the integer value to 0. The second macro provides a way to have the variable initialized to a custom value when the game starts.

The macros are simply responsible for creating an instance of the ConsoleVar class. In the class constructor we add the instance to a static linked list in order to later be able to enumerate all console variables with ease. The variable name is conveniently the same as the string representation in the debug console which results in straightforward and simple code.

Given the static linked list used to keep track of all available console variables, it is mandatory that the macros are used in the global namespace to create static variables with the same life span as that of the application. Adding a console vari-

```
CONSOLE_VAR(debugCam);
CONSOLE_VAR_INIT(godMode, 1);
...
if (debugCam)
{
...
}
```

Listing 5: A console variable can easily be registered with a macro. This creates both the variable needed for the code as well as the entry required for the console to recognize the variable.

able within a function would cause the object to be destroyed when code execution exits that namespace and the variable would cease to be available in the debug console. Typically this is not the intended behavior, which is why the implementation was done in this way without adressing the static namespace restriction.

Lastly, a debug feature that proved to be invaluable during development is the ability to render geometry in wireframe (figure 3.5). This mode draws the edges of all polygons without filling the polygons themselves. It is particularly useful to have this available in a project such as Ascent that utilizes procedural geometry. Since the draw data is not available on disk for inspection and, furthermore, it is randomized each time the game runs, it is incredibly helpful to be able to accurately visualize the whole level. Wireframe rendering, together with a debug camera that allows for more flexible movement through the scene, simplifies geometry debugging. Different render and camera modes such as these are conveniently implemented as dynamic toggleable console variables.

3.2 Game Layer

The game layer utilizes the various interfaces exposed by the engine layer and is responsible for the logic required to implement the features of the game. The goal for this project was not to create a visible split between the two layers such that they could be used independently. Rather, simplified development and maintainability was the main aspiration. By isolating clusters of logic into separate units with clearly defined interfaces, implementing, debugging, and extending features became easier through this divide and conquer strategy.

This section details the notable systems that belong to the game layer and contribute to making Ascent the unique prototype it turned out to be.

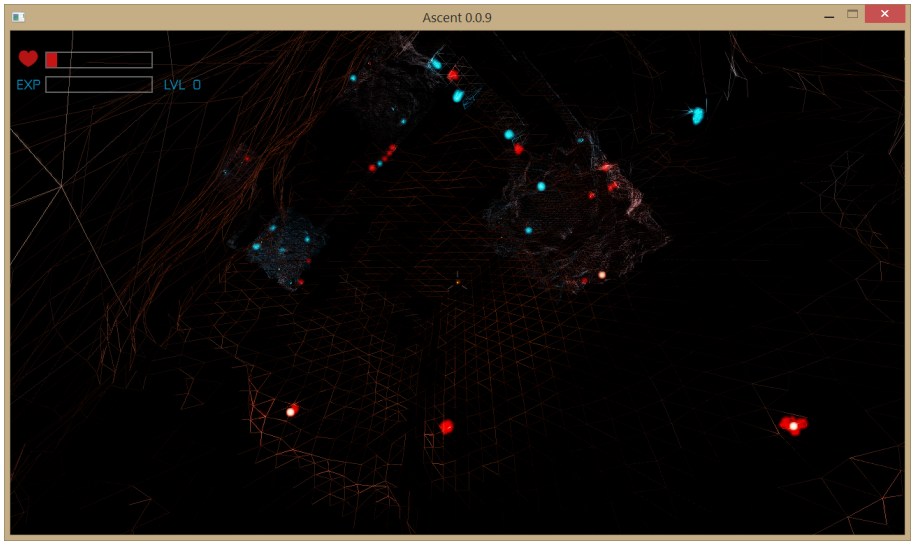

Figure 3.5: A useful debug rendering mode when dealing with randomly generated geometry is the wireframe mode. Being able to see edges clearly is invaluable when investigating procedural geometry.

3.2.1 Controls

Ascent supports both a keyboard and mouse combination as well as game pad input. With the latter, analog joysticks are supported for fine grained control of the player ship and camera.

The game camera is implemented with quaternions to allow for smooth interpolation between orientations and avoid issues like the infamous Gimbal lock [16]. Moreover, there is no artificial rotation lock. The player can rotate the camera 360° around any axis.

There are two modes available in the camera system. A debug mode implements the basic quaternion rotations and is toggleable through a console variable. This is mainly intended for development purposes as it allows the user to freely move around the game world with raw input directly controlling the camera position and orientation. Due to the nature of generic quaternion rotations, the camera can very easily gain jarring roll changes to its orientation. Naturally this does not provide a pleasant experience from a game play point of view. A game mode, enabled by default during game play, improves this camera control and loosely mimics the movement of a space ship in zero gravity.

The most significant improvement over the debug camera is an automatic alignment of the view to one of the world axes. We do this by calculating the dot product between the normalized camera's up-vector and the normal of every side of the unit cube. The largest value from this set indicates the axis closest to the camera up-vector. As long as no conflicting input is received from the player, the camera

up-vector is slowly interpolated to align with the world axis picked above. The calculated up-vector is also utilized when input from the mouse or game pad affects the yaw of the camera. This ensures yaw changes are always performed around a world axis, keeping the view more stable and not causing as much disorientation.

The pitch of the camera is easily modified by rotating along the right-vector and roll is implemented by manually rotating the camera's up-vector along the forward axis. Together with the yaw control, this provides rotation along any axis. In addition to forward, backward, and strafing movement, there are also key bindings for ascend and descend.

3.2.2 Level Generation

A level in Ascent consists of at least three and at most six rooms connected by tunnels. The first step in generating a new level, before any rooms or tunnels are created, is to pick a random direction by randomizing a normalized three-dimensional vector. This vector is used to create a directional flow for the entire level. We want the last room to be spatially far away from the first room in order for the player to get a sense of progression as he moves through the level. When dealing with procedurally generated content it is important to control the result as much as possible without sacrificing variety.

Once we have our vector for the level flow direction, we create a randomly sized room at the origin. A random vector is used to pick a point away from the origin where we would like to create the second room. If the dot product between this vector and the initial flow vector is negative, we flip the vector before picking our second room location. This ensures each time we pick a room location we stay within the half-space defined by the plane intersecting the origin whose normal is the initial flow vector. The result is a set of rooms that varies in placement yet keeps a consistent and linear flow from start to finish.

The distance between each room is chosen to be the sum of the radius of the previously created room and the maximum possible radius. The maximum radius is calculated by assuming the next room's dimensions are based on the maximum values available to the randomization process. This gives us a slight variation in distances between rooms while at the same time guaranteeing that there will be no intersections.

The first and last rooms have a single opening each, while the rest have two openings serving as tunnel connectors. A custom A* search algorithm [14] is used to connect the rooms together. A* was chosen for its virtue of always generating the shortest path between two points, which was desirable in our process of linking two rooms together. One of the modifications to the algorithm, however, limits the space traversal to world axes and prevents diagonals. The result is a multitude of sharp corners and not quite the shortest path between each room connector. An additional modification to the traditional A* algorithm is of course the ability to

move in three dimensions instead of only two.

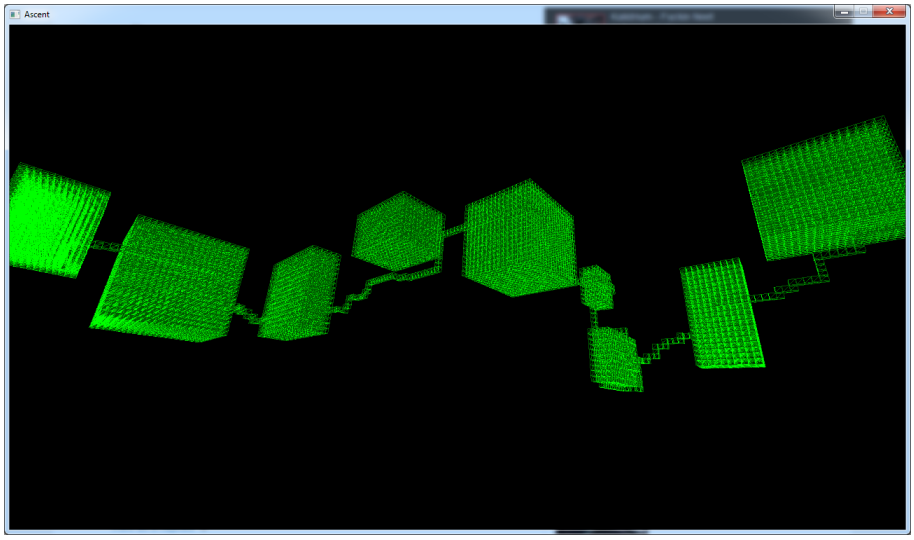

Figure 3.6: A debug render of a level portraying the rooms connected by tunnels with the help of a custom A* search algorithm.

3.2.3 Weapons

The player has a single weapon that can be upgraded by collecting blue crystals. With each level gained the cool down time between each bullet is lowered and in return the weapon can fire at an increased rate. Blue crystals, when shot, shatter into multiple parts that each give 10 experience points. The formula used for leveling up the weapon can be seen in figure 2.5. This formula is specifically designed to be open-ended, give diminishing returns at higher levels, and offer speedy progression at lower levels. This allows for an easy implementation without needing to deal with maximum capacity. The fast progression at lower levels, where most of the time will be spent by players, provides a more satisfying experience.

The logic responsible for accumulating the player's experience points, level up the weapon, and fire the bullets on input events, is all encapsulated in the weapon component.

3.2.4 Collectibles

The two types of collectibles in Ascent are identical in looks with the exception of color. Red crystals fill the player's health bar and blue crystals contribute with experience points towards increasing the level of the weapon. The crystal geometry, like all other, is randomly generated and consists of one to five crystal pieces.

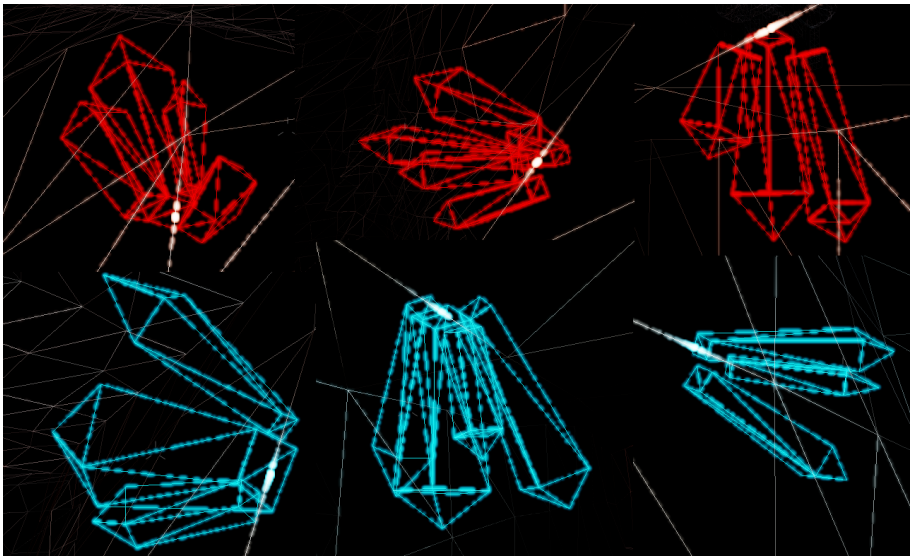

Figure 3.7: A set of randomly generated crystals shows the possible variety.

Each crystal piece consists of a cuboid with a pyramid shape attached at the top end. The width, height, and depth parameters are randomized in order to create several variations.

The crystals are procedurally placed throughout the level on the geometry of the room and tunnel walls. When a level is generated, a large set of potential crystal spawn points is generated by sampling random locations and keeping track of the coordinates and the surface normals. This data is used to place crystals, orient them correctly, and avoid intersecting geometry.

While the number of crystal pieces and their sizes vary between each crystal cluster, this is purely an aesthetic property and has no impact on game play. There is, however, a hint of randomization that does affect game play. Destroyed crystals spawn between two and ten smaller, free floating pyramid shaped parts as the cluster disintegrates. Each part gives the player ten experience points or refills the health bar by one percent, for blue and red crystals respectively. The free floating crystal parts are suspended in midair until the player gets close, at which point they gravitate towards the camera. This simplifies the gathering mechanism and allows the player to instead focus on surviving enemies and navigating the level.

Chapter 4

Future Work

4.1 Player Goals

As previously stated, the main ambition for this project was to implement a game prototype with minimal off-the-shelf library support. A fully playable, albeit very limited, game based on the classic Descent was created. One of the limitations is the lack of player goals. Beyond the essential need for the player to clear a level of enemies in order to advance to the next stage, there is no incentive to continue playing. Likewise, gathering crystals to refill the health bar and level up the weapon makes for paltry replay value.

With the base engine developed and in a stable state, additional enriching game play systems can be added with relative ease. The gathering system can be expanded to allow the player to collect additional objects, such as scraps salvaged from enemy ships. To build on this, a customization system would allow the player to use salvaged objects to improve the ship or augment the weapon. With these options, the player is presented with the opportunity to personalize the experience to a larger degree.

4.2 Polish

Polishing a game project can mean anything from fixing rare game breaking issues to adding visual effects. In Ascent, there are no known significant errors that haven't already been addressed. There is, however, a potential to enhance the prototype with usability improvements that would create a more pleasant experience.

To begin with, there is no introductory information for a new player. When the executable is launched, the level is generated and the player is placed in the game world and left alone to figure out what needs to be done. While this can be a game play mechanic in itself, a simple notification indicating what the goal is would likely improve the experience. Alternatively, a progress bar showing the

amount of enemies left to destroy in a level would both reveal to the player what needs to happen next as well as hint towards available progression.

There is currently no failure state in Ascent. When the player gets destroyed by an enemy it simply triggers a new level to be generated and the game starts again. An option would be to count the number of levels a player gets through without dying and store a persisting high score as a global goal to beat. When destroyed, the completed level count is reset and the player starts in a new level again.

Chapter 5

Conclusion

This project was implemented in C++ and the engine was based on the SDL 2.0 library. A full rendering pipeline was developed and multiple shaders were written to create the required look for the game. Additionally, a highly flexible component system is responsible for the majority of the higher level game layer with every dynamic object utilizing it.

Due to lack of resources and, in part, preferred design direction, Ascent is a fully procedural project. All content from the level and enemy ship geometry to the user interface icons is procedurally generated at runtime. A large set of parameters control these algorithms to produce varied results and unique looking enemies and level layouts for every play session.

Developing a game is a long and boundless task. Additional features can be added without limit and one can never add enough polish. Nevertheless, we set out to implement a game prototype inspired by the classic game Descent and the project turned out a success.

Bibliography

[1] Atlassian. *Bitbucket*. https://bitbucket.org. 2016.

[2] SDL Community. *Simple DirectMedia Layer*. https://libsdl.org. 2016.

[3] Microsoft Corporation. *Microsoft Visual Studio*. https://www.visualstudio.com. 2016.

[4] Erwin Coumans. *Real-Time Physics Simulation*. http://bulletphysics.org. 2016.

[5] Interplay Entertainment. *Descent*. http://interplay.com/games/descent.php. 2016.

[6] Freesound. *Freesound*. https://freesound.org. 2016.

[7] Git. *Git*. https://git-scm.com. 2016.

[8] Khronos Group. *Khronos Releases OpenGL 4.3 Specification with Major Enhancements for the Standard in Open, Cross-Platform 3D Graphics*. https://www.khronos.org/news/press/khronos-releases-opengl-4.3-specification-with-major-enhancements. 2012.

[9] Khronos Group. *OpenGL*. https://www.opengl.org. 2016.

[10] Jari Komppa. *SoLoud*. http://sol.gfxile.net/soloud. 2016.

[11] Tristan Lorach. *White paper: Soft Particles*. Tech. rep. NVIDIA Corporation, Jan. 2007. URL: http://developer.download.nvidia.com/whitepapers/2007/SDK10/SoftParticles_hi.pdf.

[12] Unity Technologies. *Unity - Game Engine*. https://unity3d.com. 2016.

[13] Procedural Content Generation Wiki. *Procedural Content Generation*. http://pcg.wikidot.com. 2015.

[14] Wikipedia. *A* search algorithm*. https://en.wikipedia.org/wiki/A*_search_algorithm. 2016.

[15] Wikipedia. *Bullet (software)*. https://en.wikipedia.org/wiki/Bullet_(software). 2016.

[16] Wikipedia. *Gimbal lock*. https://en.wikipedia.org/wiki/Gimbal_lock. 2016.

[17] Wikipedia. *MurmurHash*. https://en.wikipedia.org/wiki/MurmurHash. 2016.

[18] Wikipedia. *Phong shading*. https://en.wikipedia.org/wiki/Phong_shading. 2016.

[19] Wikipedia. *Roguelike*. https://en.wikipedia.org/wiki/Roguelike. 2016.