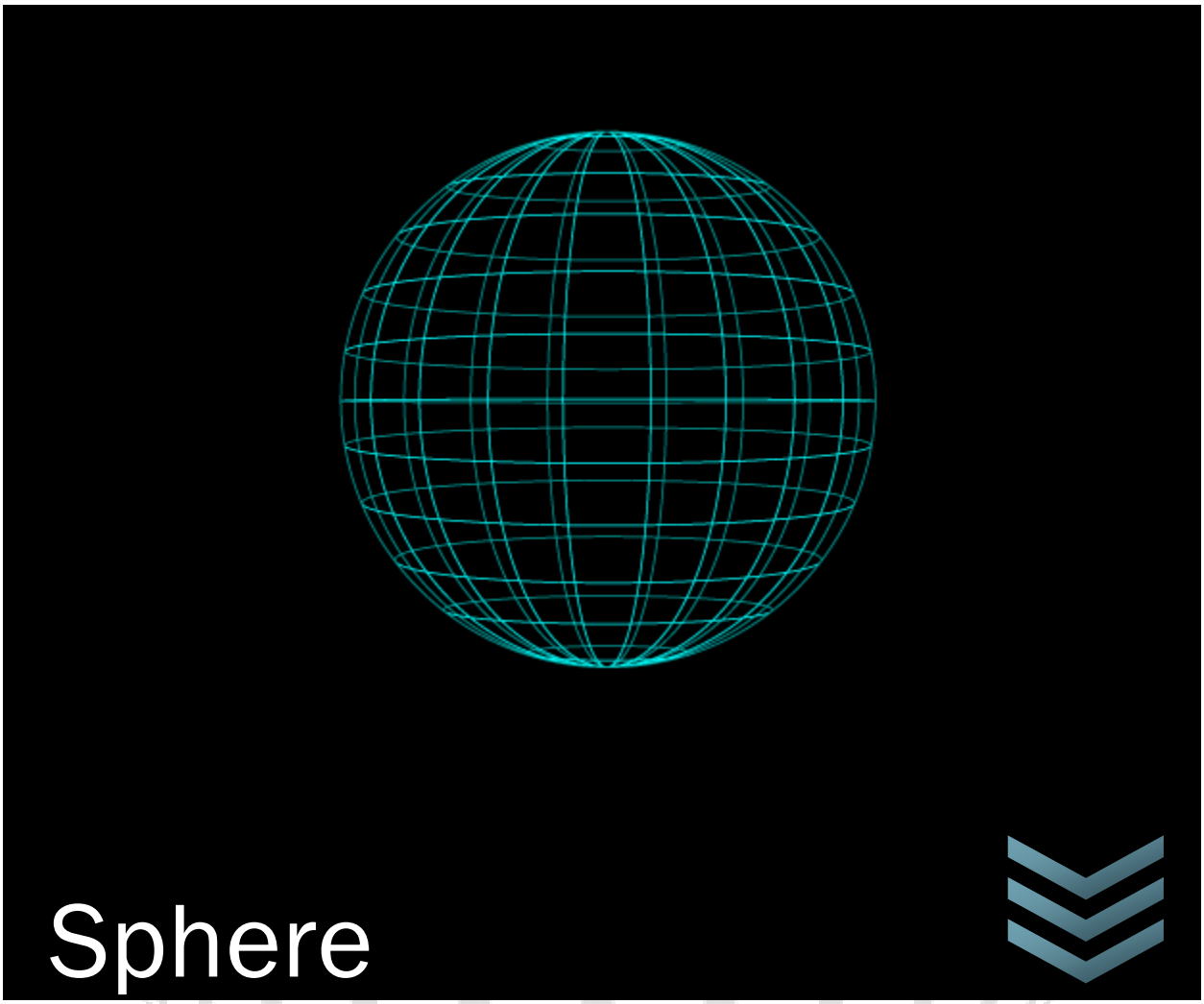

Sphere

Alexei Andreev

Kevin Berg

Joe DiGiovanna

Marc Holden

Robert Lach

Michael So

Paul Taraszka

Team Sphere

Documentation

THIS PAGE INTENTIONALLY LEFT BLANK

1. Table of Contents

2. Overview

Sphere is a single player game, where the player guides a sphere through a series of challenges. The game features 3D graphics and a third person view, centered on the avatar. The game has a simple background story, a single objective, and a series of puzzles and enemies. The player will have to use available abilities to solve the puzzles and eliminate or avoid the enemies. The game requires the player to have quick reflexes and good spatial reasoning.

Overall, the game is meant to be challenging on multiple levels and, most of all, fun. The player is not expected to fail on the first couple of attempts. The pace of the game will vary from fast to slow, based on what the player is doing. A player of any age should be able to enjoy the game, however it will require a certain mastery to complete it.

The Sphere game is written in C++ using several libraries, including OpenGL, OpenAL, SDL, and several others. The code base in separated into several main technical components: video, audio, input, gui, game, and testing. Game component consists of AI, cameras, controllers, enemies, items, projectiles, terrain, worlds, and several others. The game control flow can be divided into several phases: initialization, game, and tear down.

The overall game design is very traditional. The code base and the game are divided into several predictable components. Each component is designed to be responsible for only one task or feature, and be independent of other components. This leads to modular design and cohesive code structure. The performance issues are also well known and addressed in the most straightforward fashion.

3. Process

Team Sphere uses an iterative incremental agile software development process called Scrum. The Scrum method is popular because it is easy to learn and it is very adaptable. By separating the involved people into categories, Scrum keeps the responsibilities transparent, which helps everyone to know exactly what is expected of them. Scrum assumes that all requirements cannot be known in advance and that the product goal will change during the development process. Thus, each Sprint the Team is able redefine its view of the project, adapt, and set new goals. This keeps the entire process flexible in the face of constant changes.

The Scum process was created from previously existing ideas about software development, which came from several key people. DeGrace and Stahl first coined the term Scrum. Ken Schwaber and Jeff Sutherland were one of the first to actively use the method in their respective software developing companies. Lastly, Sutherland and Schwaber took and merged the existing ideas and practices associated with Scrum and wrote them up in the book called Agile Software Development with Scrum.

Scrum development process divides the participant into several key roles. The main role division is into chicken and pigs, which comes from a joke:

A pig and a chicken are walking down a road. The chicken looks at the pig and says, "Hey, why don't we open a restaurant?" The pig looks back at the chicken and says, "Good idea, what do you want to call it?" The chicken thinks about it and says, "Why don't we call it 'Ham and Eggs'?" "I don't think so," says the pig, "I'd be committed but you'd only be involved."

Individuals who are fully committed to the product and are actively participating in its creation are called pigs. They consist of one Scrum Master, one or more Product Owners, and the Team.

Scrum Master is the person who is certified in Scrum and usually has previous experience with the development process. This position is of a leader or a manager, but of a consultant. He or she is there to help the development team deliver their project with minimal hassle and in timely manner. This person makes sure the Team is following the process accurately and, if there are any problems, Scrum Master will follow Scrum process to suggest the best solution. He or she is also responsible for removing any distractions or barriers that may interfere with the Team's goal of delivering a quality software product on time.

Product Owner represents the customers who ordered and will use the finished software product and ensures the quality of the final product. This person makes sure the software is

up to the set standard and minimum requirements are met. To ensure this happens, Product Owner writes user stories and adds them, in order of importance, to the Product Backlog. This Backlog is a formal definition of the final product and is frequently consulted to make sure the product development is on track and making progress.

The Team carries most of the burden in delivering the finished product in good quality and on time and consists of the people who actually work on implementing the product. The Team is usually made up of approximately seven people with diverse set of skills, including developers, designers, testers, and additional roles, depending on the type of software.

Since our group's game project is relatively small in scope and we don't have many people working on it, we have many overlapping roles. We don't have a Scrum Master, since none of us is certified. However, all of us know and understand the process, so we all share the responsibility. We consider our TA to be the Product Owner, since he reviews our game on periodic basis and offers suggestions. However, the entire Team shares the responsibility for creating user stories and Product Backlog.

The chickens consist of Users, Stakeholders, and Managers. Users are the people who will end up using the final product. If they have any concerns or recommendations, they can consult the Product Owner, who might take their comments and, in turn, pass them to the team. Stakeholders consist of customers, vendors, and other people who are in some, usually financial, way invested in the project. Managers are responsible for organizing the involved personnel, setting up meetings, and providing the resources necessary for the project's success.

Our group doesn't have many chickens, since the scope of our project is really small. Our users are the players who will play the game, but we don't have any stakeholders, since there our project has financial aspect. Our TA is also partly our manager, since he provides us with guidance and help.

The whole Scrum software development process is divided into Sprints. A Sprint it usually somewhere between two to four weeks in length and has its own list of goals. Before the Sprint starts the Scrum Team, aka pigs, gather for a meeting and decide on the features and goals they would like to have done by the end of the Sprint. Product Owner has an important role during this meeting, since it's primarily his/her responsibility to come up with a sound list of Sprint goals. These usually come from the Product Backlog, which contains all the user stories and requirements needed for the final version of the software product, as specified by the Product Owner. Once the Team and the Product Owner discuss the assigned goals and make sure both are comfortable with the work required of them, they

compose the final list of Sprint goals and add it to the Sprint Backlog. This Backlog will remain unchanged for the duration of the Sprint.

After the meeting, the Team gathers on its own and decides on the best way to solve the given Sprint Log tasks. This involves dividing up the work, assigning tasks, and making sure all the task dependencies are clear, so the Team can go through the Sprint in a smooth manner. After each Team member has their list of Sprint tasks, they break up each task into small set of easily quantifiable goals. Each goal should be clear and should only take a couple of hours. Once all the goals for the task are completed, the task can be counted as finished.

Since we only have one semester to finish our project, we decided to go with smaller Sprints, or, as we dubbed them, Super-Sprints. Each one is only a week long. This also means all the task are smaller, so they do not have to be separated into goals. During each meeting before a new Sprint, we look at the user stories and consider what we have left to do. We then compile a list of required tasks and assign each task to a member.

Usually the Team keeps all the tasks and goals in publicly accessible place in order to track the progress. This can be as simple as a white board with sticky notes or as complicated as a specialized software program. After a Team member broke down each task into a set of goals, he or she should predict how long each goal will take. This is the predicted time and allows the Scrum Master or the software program to compute the total number of work hours planned for the current Sprint. As the Team member works on the task, he or she updates the amount of time left for the task. Since unexpected things happen quiet often, that time can at some point be higher than originally predicted . Over time, however, the total number of hours left should go down. A graph of the remaining time is called the Burndown chart. Ideally, it should reach zero right at the end of Sprint.

Likewise, we also predict the number of hours it will take us to complete the given tasks and put them on a wiki page. Once a task is done, we write down the number of hours it took us to complete the task. This way we can predict more accurately and can notice if we tend to over- or under-estimate the amount of work. Since our Sprints are short, we do not have a Burndown chart.

Every day the Team meets with Scrum Master during, what is called, a Daily Scrum meeting. Each Team Member discusses what goals they've completed and what they are planning to do next. The other Team members are free to ask questions and give suggestions. A team member can also request help from Scrum Master if there is some problem or dependency, which prevents the Team member from doing their work.

Sprints usually follow one another in succession until the project is done, with meetings in between. During the end-of-sprint meeting The Team holds discusses the Sprint: what went wrong and what went right. This will allow the Team to adapt better for the upcoming Sprints. The Team also hosts a demonstration for the stakeholders and users. The Teams shows off what they've accomplished, answers questions, and takes comments. After that, a new Sprint meeting is in order and a new Sprint begins. Thus the cycle continues until the project is completed.

When a Sprint ends normally all unfinished tasks, if they are still necessary or relevant, are carried over to the next Sprint. Occasionally, something extraordinary will happen and the Sprint will have to be interrupted before its due time. In this case, Scrum Master will advise the best course of action and the Team might have to start a new Sprint with the new, redefined tasks.

Since the team did not accomplish much work from day to day, we do not have daily meetings. Instead, we just use the Sprint Review meeting to go over our accomplished goals and encountered problems. Every other Sprint we have a Milestone, where we have to present our project to the owners.

4. Requirements and Specifications

The requirements and specification for Sphere can be divided into two main areas: game design specifications and technical specifications. The separation of the two main areas is intrinsic to game development.

4.1 User Stories

1. Player should be able to control the avatar via keyboard and mouse. The avatar should respond by moving. A camera should respond by following the avatar.
2. The avatar should be able to interact with the world though collision detection and gravity.
3. The avatar should be able to acquire different abilities in its closed form. These abilities should be visible. The abilities should be used in the world to achieve goals.
4. The avatar should be able to pick up items and power-ups.
5. The avatar should be able to transform into the open form. Its visual appearance should change according to the different attachments the player collected.
6. Player should be able to interact with the world when the avatar is in the open form. Avatar's attachments will dictate what kind of interaction the player will be able to have.
7. The world should be populated with enemies. The enemies should have some sort of AI. The enemies should be able to interact with the world through collision detection and gravity.
8. Player should be able to use avatar's abilities and attachments to interact with enemies.
9. Player should be able to complete a level by meeting a certain goal. After completion of all levels, the player wins the game.

4.2 Game Design Specifications

Sphere is a single player game, where the player guides a sphere through a series of challenges. The game features 3D graphics and a third person view, centered on the avatar. The game has a simple background story, a single objective, and a series of puzzles and enemies. The player will have to use available abilities to solve the puzzles and eliminate or avoid the enemies. The game requires the player to have quick reflexes and good spatial reasoning. The rest of this document will further describe the specifications for background story, avatar design, player controls, world, puzzles, enemies, and other various gameplay elements.

The game features a simplistic storyline to help the player immerse themselves in the game. The story is told when she starts the game: You, a technologically advanced sphere, is sent on a mission to cover up alien tracks on Earth. The first mission is to blow up the pyramids, which, as everyone knows, are actually alien spaceships. Your goal is to make it into the pyramid and detonate yourself. On the way you will acquire different abilities, which you will use to overcome enemies and puzzles.

Your outer sphere shell is made of simple alloy of alien design. As you progress through the game, you will be able to collect and use more shells: rubber, glass, and magnet. Rubber will let you bounce without losing energy; glass will decrease friction, thus increasing movement speed; and magnet will allow you to attach yourself to metal surfaces. While the avatar is a sphere, it is in the closed form. Besides shells, aka. closed form abilities, the player can also collect weapons and other attachments. While in the closed form, the avatar can open up, where the sphere is separated into hemispheres connected by an axle. This form is called the open form. It's similar to a car's axle . At the center of the axle is the attachment point. If the player collected any open form attachments, they will appear and they player will be able to use them. For example, if the player picked up a gun, she would be able to shoot it while in the open form. Some puzzles can only be solved in open form. Enemies, for example, can only be killed with a weapon, and to use a weapon the avatar has to be in the open form. The player has to switch between the two forms in order to complete the game.

The game controls depend on what form the avatar's active form. In the closed form, the player controls the direction of the roll and the camera direction. The speed and maneuverability varies depending on the surface the avatar is on, as well as the active closed form ability. In the opened form, the player controls only the direction the avatar is facing; the camera faces the same way. To move, the player needs to have appropriate attachments. Legs, for example, allow the player to jump and walk, but to use them the player needs to collect the legs attachment.

The game is separated into several levels, each one carefully crafted for predetermined player experience. To progress through the entire game, the player will have to solve each level. I will mention only the first level here: the desert. However other levels follow a similar progression. The player starts out with no attachments or closed form abilities. From the start, she is presented with simple navigation puzzles. Once the player acquires a closed form ability, the following puzzles will require her to use it. Generally, there is only one way to progress through the level and it's ensured by the overall level layout and strict order in which the player acquires the abilities.

Each level is filled with enemies the player has to overcome. Enemies move and react to the player, and most can attack and damage the player's health. The player needs a weapon in order to destroy the enemies. Some enemies are vulnerable to only one kind of weapon. For example, the deadly swarm can be destroyed only with a spray gun. Some enemies cannot be destroyed at all and must be avoided. It is up to the player to explore the possibilities and decide the best course of action.

There are several other gameplay elements the player will encounter during the game. One is colored keys, which are used to opened doors of matching color. There are several others: damage multiplier, invulnerability, and armor. Each affects the avatar in different beneficial ways.

The overall visual game appearance is meant to be simplistic. None of the models, except for puzzles, are textured. They use single colored materials and are affected by lighting. The ground and the puzzles are textured in bright colors, using similar color scheme to be consistent.

Overall, the game is meant to be challenging on multiple levels and, most of all, fun. The player is not expected to fail on the first couple of attempts. The pace of the game will vary from fast to slow, based on what the player is doing. A player of any age should be able to enjoy the game; however it will require a certain mastery to complete it.

4.3 Technical Specification

The Sphere game is written in C++ using several libraries, including OpenGL, OpenAL, SDL, and several others. The code base in separated into several main technical components: video, audio, input, gui, game, and testing. Game component consists of AI, cameras, controllers, enemies, items, projectiles, terrain, worlds, and several others. The game control flow can be divided into several phases: initialization, game, and tear down. The rest of this document will discuss each phase and component in more details, as well as mention their importance and some technical optimizations and concerns.

The initialization phase occurs first when the game is run. The components are setup in the following order: video, audio, gui, and game. Once all components are initialized, the game phase starts. During this phase the game is repeatedly updated and rendered. When the user decides to quit the game, the components are torn down in the opposite order they were initialized. During the game phase, each update and render cycle makes up one frame. The number of frames per second is a very important quantity and is usually the main target for optimization.

The video component is the most important one, since it's directly involved in user's visual experience of the game. It includes SDL and OpenGL. SDL is used to setup the window or full screen environment, enable multiple capabilities: rendering, timing, window caption, and double buffering. OpenGL is used to render 3D graphics and to load related resources, like textures and shaders. There is another library, GLEW, that is used to open up OpenGL's extended functionality. It allows the programmer to use the new functions and paradigms which exist in the newer versions.

The audio component is important as well, since sounds give useful clues to the player. OpenAL library is very limited by itself, so we use it in conjunction with OGG and Vorbis libraries, which allow us to play .ogg sound files. The sounds are buffered on demand, which allows the system to have very low memory consumption. The library also allows for 3D sound positioning, as well as velocity and direction parameters. This allows the player to more easily immerse themselves in the game.

The input component is important for user's experience with the game's controls. It's implemented as an extension of SDL, called SDLInput. It allows us to get the immediate mouse and keyboard states. We created a wrapper for it to allow a programmer to check at any time if the key is currently pressed or released, and if the key has been pressed or released during the current frame. The programmer can also retrieve the current location of the mouse or the amount of pixels the mouse moved since the last frame; whether a mouse button is down or up, or has been clicked or released this frame. This duality allows for great flexibility in coding.

The GUI component is necessary to indicate to the player the current status of the avatar, the game, and allow the user to interact with the game on a meta level. The gui frameworks

consists primarily of images, buttons, and text objects. All these objects have a standard implementation and functionality.

The testing component is necessary to ensure a minimum level of quality and functionality. It's primary use is to help programmers. It uses `CppUnitLite` framework, which has a very simple set of functions that allow programmers to create unit tests. Our game has unit tests for all the major components, including some game components. It's important to note that it's very hard to create unit tests for a game, because a lot of information is visual and cannot be tested by a program.

The last and the most important component is the game itself. Its code is designed to be essentially a one-to-one map between the gameplay world and the corresponding classes. There is a separate class for each kind of enemy, item, projectile, terrain, and attachment. Each enemy has an AI class associate with it, which controls the enemy's behavior, including movement and attack. The avatar has a similar class, but instead of AI, it's the player who is controlling it. The avatar controller checks the user input and makes the avatar respond to it in an appropriate fashion. There are also classes for abstract objects like camera, bounds, the world in general. The game code is designed to be very modular and efficient.

The overall game design is very traditional. The code base and the game are divided into several predictable components. Each component is designed to be responsible for only one task or feature, and be independent of other components. This leads to modular design and cohesive code structure. The performance issues are also well known and addressed in the most straightforward fashion.

5. Architecture and Design

5.1 The Metaphor of Sphere

Sphere is like an existential journey as a polar bear, hopping from ice block to ice block, occasionally slipping but ultimately reaching safety and achieving one's goals.

5.2 Framework Architecture

The two primary frameworks used for this game are: `OpenGL` and `SDL`. `SDL` is used to set up the window (including full screen mode if applicable), control resolution, manage timers, and provide access to user input. For more information see: http://www.libsdl.org/

Our game has a thin wrapper around `SDL` to: 1) make it conform to Object Oriented programming model and 2) abstract the required functionality. The wrapper also provides several services, like selecting the best resolution and frame buffer parameters.

OpenGL is used primarily for graphics and other related functionality. This includes: matrix manipulation, texture loading and handling, and the entire rendering pipeline: lighting, depth testing, texturing, culling, fog, double buffering, and perspective. For more information see: http://www.opengl.org/

Both frameworks are initialized when the game starts and released when the game terminates.

5.3 World and Game Mechanics

5.3.1 Game Objects

All of the objects in Sphere are extended from the SphObject class. Each object in the game has its own class, and most objects are one of the two main extensions of SphObject. The in-game avatars for both the player and the enemies are SphModel. The interactive non-animate objects in the world are SphProps. The light in the world is a SphLight. Objects that have a visual representation in the world will have a SphMesh associated with them.

SphModels in the game will be SphAvatar, SphEnemy or SphCrate. Each SphModel has its own Update function which describes its behavior in the world. There is only one SphAvatar at any given time: the avatar that the player is currently controlling. The SphAvatar will interact with other objects in the game through events such as collisions. Enemies in Sphere are SphEnemy. Enemies populate the world to hinder the player's progress. The enemy's characteristics are contained within each individual enemy subclass. Adding a new enemy type is accomplished by creating a new enemy class that extends SphEnemy. The last type of SphModel is a SphCrate. SphCrates are placed in the world to hold items that the player will pick up.

SphItem, SphProjectile, SphMovingProp and SphQuicksand extend SphProp. Items located throughout the world that the avatar picks up are SphItems. SphItems cannot be placed in the world as items must serve a specific purpose, thus each actual item must be a subclass of SphItem. This includes SphMedpack which are the medpacks in the game that restore health to the avatar, SphKey which are the keys to open crates, and SphAbilityItems which are the items that give the avatar its open and closed form abilities. Projectiles that the avatar shoots are SphProjectiles. All of the projectiles characteristics including range and damage are contained here. Adding a new projectile type is accomplished by creating a new projectile class that extends SphProjectile. SphMovingProps are SphProps that move around the level. Both SphProjectiles and SphMovingProps have Update functions that describe their behavior in the world. This includes the SphMovingPlatform subclass, a platform that the avatar rides to solve a certain puzzle. SphQuicksand is a prop that is placed throughout the world to provide a natural barrier that the player cannot cross. See Figure A.

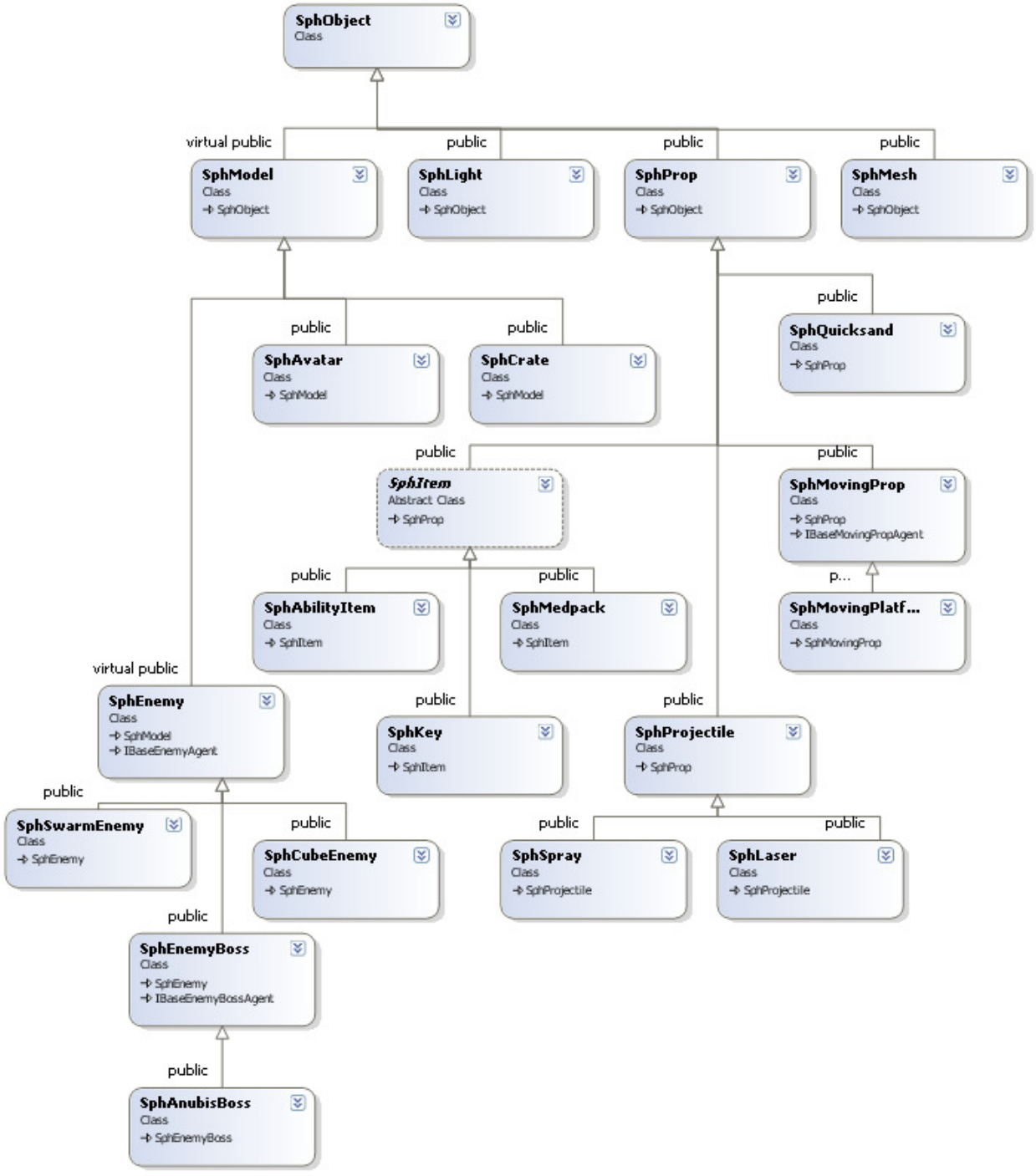

Figure A (UML Diagram of Object Classes, Team Sphere)

5.3.2 World Class

Levels in Sphere are worlds. These worlds are part of the `SphWorld` class. Each individual world is an extension of `SphWorld`. Each world contains all of the `SphAvatar`, `SphObjects`, `SphPuzzles`, `SphEnemies`, etc. and places them in their correct locations in their `AddPuzzles`(), `AddItems`(), `AddEnemies`(), `AddOther`() and `AddLights`() functions. For example, `SphDesertWorld` is a desert/savannah world that adds content relevant to that world, including `SphSwarmEnemys` and desert based `SphPuzzles`.

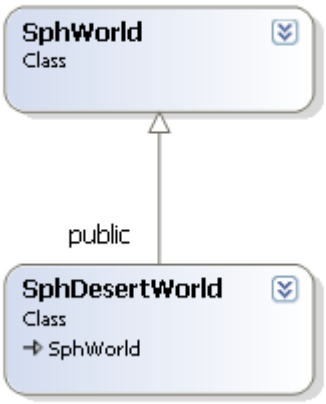

Figure B (UML Diagram of World Classes, Team Sphere)

5.4 Input

Input is handled by the class "SphAvatarController". The design of SphAvatarController was heavily influenced by use of the SDL extension, SDLInput. The controller is an object that contains a pointer to an avatar and two animation controllers (m_controller, m_legsController) as seen in the following constructor.

```
SphAvatarController::SphAvatarController(SphAvatar* avatar) :
m_controller(3), m_legsController(4){
      this->m_avatar = avatar;
      this->SetupController();
      this->m_controller.SetModel(avatar);
      this->m_legsController.SetModel(avatar);
}
```

Figure C - (Avatar Controller Code Sample, Team Sphere)

After the object is set up, SDLInput makes it very easy to catch all of our keyboard and mouse actions. Inside of each sdlInput event, there is code to perform the necessary actions, as seen below.

```
if(g_sdlInput.IsKeyDown(SDLK_w)){
            Multiply(forward, moveFactor);
            m_avatar->AddForce(forward);
            transition = C_ForwardWalk;
      }
else if(g_sdlInput.IsKeyDown(SDLK_s)){
            Multiply(forward, -moveFactor);
            m_avatar->AddForce(forward);
            transition = C_ForwardWalk;
      }
}
```

Figure D - (SDL Input Case Code Examples, Team Sphere)

The controls are assigned to be as familiar to the user as possible. This was a simple task for implementing keyboard controls, however, mouse controls are a tad more complex. The user starts out with the mouse controlling the camera. It is fairly standard camera control and allows for zooming in and out. When a user right clicks with the mouse he surrenders his camera control for a cursor that can be used to interact with certain elements of the GUI, as shown.

```
if(g_sdlInput.MouseButtonUpEvent(g_sdlInput.s_rightMouseButton)||
   g_sdlInput.KeyUpEvent(SDLK_g)){
            g_sdlInput.SwitchMouseControl();
}
```

Figure E - (SDL Input Mouse Button Even Cast Example, Team Sphere)

For a list of detailed controls please see *Appendix A: Game Controls*.

5.5 Graphical User Interface

The GUI interacts directly with the avatar and changes as the avatar is updated. When the avatar gains new abilities, the heads-up display is called to display the new ability the avatar has acquired. If an ability has been acquired for the first time, a tooltip is briefly displayed which explains the usage and properties of the ability.

```
void SphAvatar::AddClosedType(SphAvatar::ClosedType type){
	if(!m_cfa[type]){
		m_cfa[type] = true;
		g_game.m_hud->EnableCFAButton(type);
	}
}
void SphHUD::EnableCFAButton(SphAvatar::ClosedType type){
	if(!m_cfaTooltipShown[type]){
		m_cfaTooltipShown[type] = true;
		m_tooltipActive = true;
		m_tooltipTimer = s_displayTime;
		RemoveTooltips();
		g_gui.AddGuiObject(m_cfaTooltip[type]);
	}
	g_gui.AddGuiObject(m_cfaButtons[type]);
}
```

Figure F - (GUI Button Implementation Code, Team Sphere)

Open form abilities are displayed along the bottom of the screen while closed form abilities are displayed along the left side. The closed form abilities also double as buttons which change the avatar to the selected form when clicked. Other elements of the GUI such as health and current closed form are displayed in a static location in the bottom left corner while items acquired, such as keys, are displayed in the bottom right corner.

```
void SphHUD::UpdateHealth(){
	_itoa(m_avatar->GetHealth(), m_health->m_text, 10);
}
```

Figure G - (GUI Item Update Example Code, Team Sphere)

As the properties of the avatar change, the HUD reflects those changes immediately so the user accurately knows the state of the avatar.

The GUI also interacts directly with the game and provides the user with options such as pausing, restarting a level or quitting the game. When the main menu is brought up, the game is paused and the user can resume, restart the level or quit. The game can also be paused through the developer console.

```
void SphGame::Update(float seconds){
	static bool musicPaused = true;

	m_hud->Update(seconds);

	if(!(m_hud->MainMenuFocus() || m_hud->ConsoleFocus())){
		m_camera->Update(seconds);
		m_world->Update(seconds);
		m_skybox->update(m_avatar->GetPosition(), 160);
		m_terrain->update(m_avatar->GetPosition(), m_world);
	}
}
```

Figure H - (Game/GUI Update Interaction Example, Team Sphere)

The console allows the developer to directly change the properties of the avatar by adding or removing abilities, changing the health or position of the avatar, and changing the way the game is displayed on the screen.

5.6 Models and Animations

Sphere game uses a proprietary custom made model, mesh, and animation formats. The models are made in Blender and are exported using custom Python scripts. The game can load the exported files. To explain the underlying structure: every model consists of a skeleton and a set of actions; a skeleton is made of bones; every bone can have a mesh attached to it; every action manipulates bones. Bones and actions are stored in the model file. Meshes are stored separately. Meshes can be loaded and displayed by themselves, without using a model. For example, all puzzles are simply meshes. Consult Figure I.

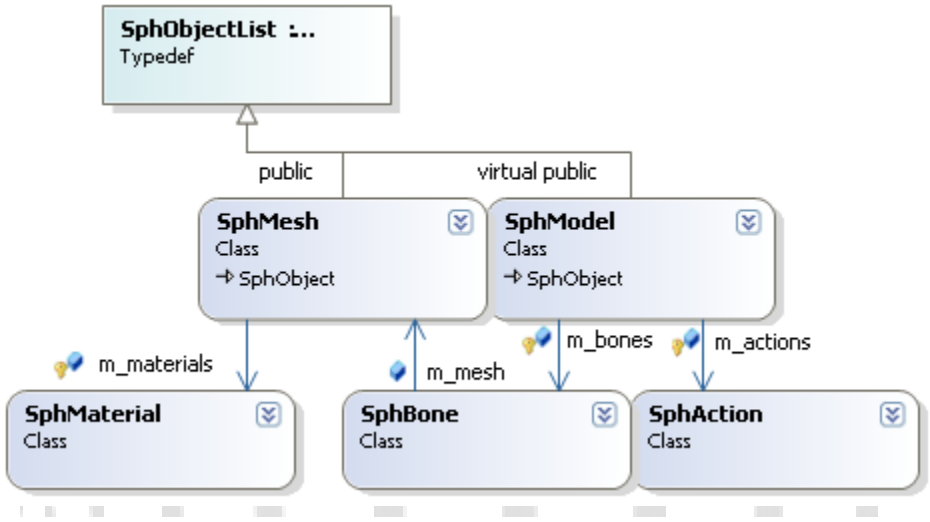

Figure I - (UML Diagram of Model/Mesh/Material classes, Team Sphere)

5.7 Physics and Collision

Every object has an abstract collision bound (although not all objects use it). We have several types of bounds: point, spherical, and tight. Point bound approximates a point and is used for projectiles. Spheric bound is described by a point and a radius and is used for items, avatar, and enemies. Tight bound is described by a set of triangles and is used for ground and complex meshes, like puzzles. Each object can instantiate of a bound most appropriate for it.

The collision detection is handled by the world. Each object can request to be collided with all collidable objects. The world will process the collisions and return a list of them, where each member is a `SphCollisionInfo`, which contains information about the collision.

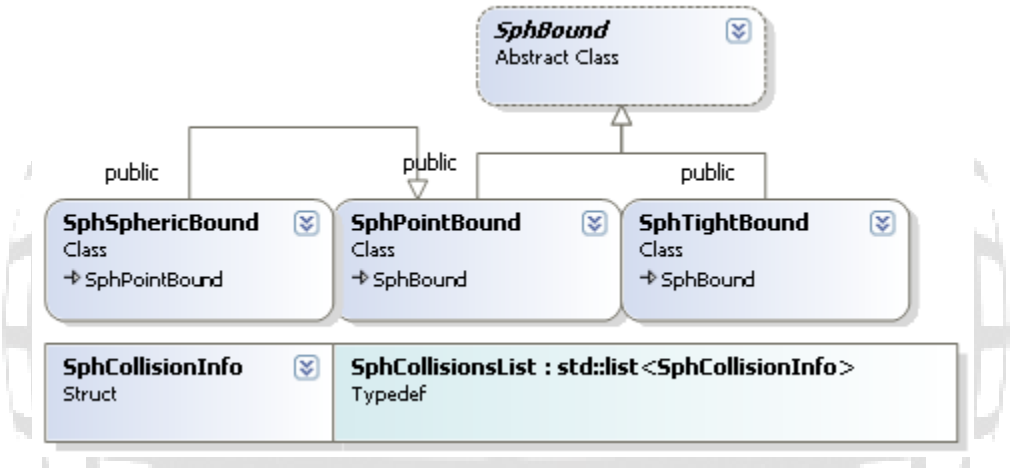

Figure J - (UML Diagram of Collision Bound Types and Interaction, Team Sphere)

5.8 Artificial Intelligence

Enemies and other moving objects populate Sphere's game world, and each has a unique Artificial Intelligence (AI) controlling them. The AI in Sphere is a simple Finite State Machine (FSM) model, with the FSM's states representing the currently behavior of the AI-controlled object, and with deterministic transitions between the object's behavior states.

The major components of the AI system are the Agent, the States of the FSM, and the Entity. Every Entity, which may be an Enemy or a moving platform, contains an "Agent", the object authorized to act for the Entity. When the Entity updates, it causes its Agent to update, which in turn executes the logic of its current state, deciding whether to perform the current State's action or to transition to another State. The Agent keeps track of the current State and acts as an interface between the FSM and the Entity, controlling the Entity's physical actions.

Moving platforms are an example of one of Sphere's AI-controlled entities. Essentially, the platform may be in one of two states: either it is "moving" or it is "idle". An example of the moving platform's FSM, which updates on the object's "Update" action, is detailed in Figure K. As one may see, very simple factors determine the transitions and executions of each state, such as the "time spent idle versus the time to remain idle" and "whether the object has reached its set destination".

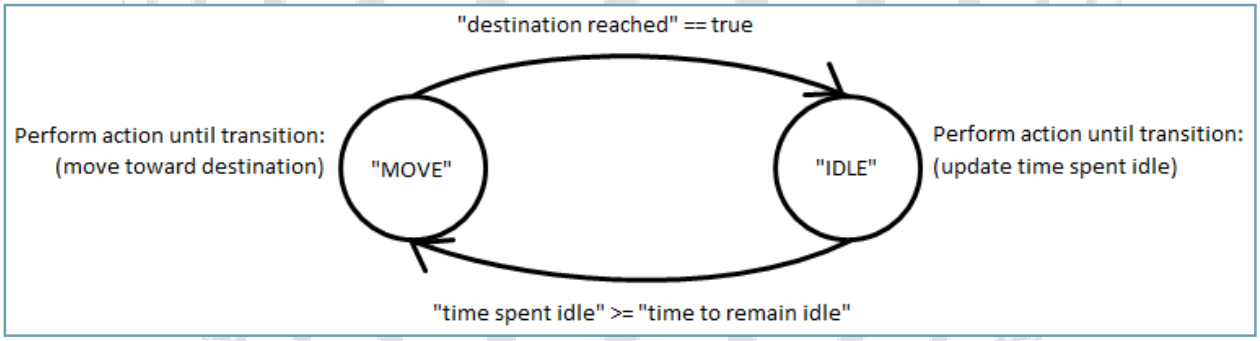

Figure K - (Diagram of Example Finite State Machine, Team Sphere)

5.9 Audio and Sound Effects

Sphere utilizes the OpenAL framework for interfacing with hardware. There are allocations for eight audio channels which are dynamically accessed and freed. For each iteration through the game loop, the audio framework iterates through each active audio channel and checks if the audio is still playing. If the audio stopped playing or was explicitly stopped by the user, the audio channel is freed and flagged for possible allocation.

```
//Update OpenAL system.
void SphOpenAL::Update(){

	for(int n = 0; n < s_musicCount; n++){
		if(!m_music[n].IsActive()) continue;
		bool stillActive = m_music[n].Update();
		if(!stillActive) m_music[n].Free();
	}
}
```

Figure L - (Audio Update Code Example, Team Sphere)

The audio system is divided amongst two classes, `SphMusic` and `SphOpenAL`. `SphMusic` is dedicated to each individual audio file. Each `SphMusic` object has positional data associated with it which allows the game to connect physical objects within the world with their associated sounds. The system allows for setting positions and velocities, which allows for automatic panning and Doppler effect. Consult Figure M.

```
//Set the position of the music m_source in global coordinates.
void SphMusic::SetPosition(cfloat* position){
	alSourcefv(m_source, AL_POSITION, position);
}

//Set the velocity of the music m_source.
void SphMusic::SetVelocity(cfloat* velocity){
	alSourcefv(m_source, AL_VELOCITY, velocity);
}

//Set the direction of the music m_source.
void SphMusic::SetDirection(cfloat* direction){
	alSourcefv(m_source, AL_DIRECTION, direction);
}

//Set the rolloff factor for the music m_source.
void SphMusic::SetRolloff(cfloat rolloff){
	alSourcef(m_source, AL_ROLLOFF_FACTOR, rolloff);
}
```

Figure M - (Audio Property Code Example, Team Sphere)

Each sound file is allocated and played with a single function which exposed globally which allows any class to play a sound. This property allows for any class to play a sound which makes associated sounds with actions within the game extremely straightforward.

```
// Will open a .ogg file and play it. The audio will loop if
loopForever is true.
int SphMusic::OpenAndPlay(cchar* filePath, bool loopForever){
	if(ov_fopen((char*)filePath, &m_oggStream) < 0) return 1;

	vorbis_info* vorbisInfo = ov_info(&m_oggStream, -1);
	m_format = (vorbisInfo->channels == 1) ? AL_FORMAT_MONO16 :
AL_FORMAT_STEREO16;
	m_rate = vorbisInfo->rate;

	alGenBuffers(N_PAIR, m_buffers);
	alGenSources(1, &m_source);

	alSourcei(m_source, AL_SOURCE_RELATIVE, AL_FALSE);
	SetRolloff(0.0f);
	SetPosition(g_zeroes);
	SetVelocity(g_zeroes);
	SetDirection(g_zeroes);
	if(StartPlaying(loopForever))
	{
		return 0;
		Update();
	}
	return 1;
}
```

Figure N - (Opening and Playing Accessor Code Example, Team Sphere)

5.10 Testing Framework

Sphere uses the CppUnit Lite Testing Framework, written by software developer Michael Feathers, to run Test Suites that test main portions of the game's engine for computational accuracy and stability. The Test Suites of our system, written entirely in C++, test many of the important components of Sphere's framework and classes, including its OpenGL, Sound, SDL, SDL Input, GUI, Collision, Math, and Item classes. Test cases are written by implementing methods with the descriptive method name "TEST(test_case, test_suite)" (replacing "test_case" with the test case name, and "test_suite" with the test suite name), which makes understanding the functionality of each test case a simple task.

Figure O demonstrates one of our OpenGL Test Suite's test cases, which tests the initialization of OpenGL (normally called upon starting the program). The test itself is written in the TEST(init, OpenGL) method ("init" is the test case name, "OpenGL" is the test suite name). The other function, "check_openGL_error()" is not unique to the testing framework, and only serves as a specialized check for OpenGL errors.

```
int check_openGL_error()
{
	//CHECK FOR OPENGL ERRORS
	GLenum errCode;
	const GLubyte* errString;
	if((errCode = glGetError()) != GL_NO_ERROR){
		errString = gluErrorString(errCode);
		return (int)errCode;
	}
	else{
		return 0;
	}
}
// Tests for OpenGL initialization
TEST(init, OpenGL)
{
	printf("-Running OpenGL Initialization Test - ");
	CHECK(!g_sdl.Init(0)); //INITIALIZE SDL WINDOW FIRST
	CHECK(!g_openGL.Init());//INITIALIZE OPENGL
	CHECK(!check_openGL_error());//CHECK FOR OPENGL ERRORS

	//make sure width and height are set
	CHECK(g_openGL.GetScreenWidth() != 0);
	CHECK(g_openGL.GetScreenHeight() != 0);

	g_sdl.Quit();
	printf(" DONE \n");
}
```

Figure O - (Unit Test Code Example, Team Sphere)

5.11 Terrain and Environments

The environment in the game Sphere is comprised of a skybox and terrain and the terrain's subcomponents: the trees and grass. Because it can be difficult to setup the grass and trees correctly for the terrain, we also developed a small level editor to make code generation a simpler task. Since the grass and trees are linked to the terrain, we focused more time on optimizing them and making sure we had good cohesion between all components.

The terrain is the most complex and the central most of the environmental components. In order to meet predetermined performance and aesthetic requirements for the game, we implemented several key features. The first feature we added was detail texturing. Although it comes at a relatively low cost to the frame rate, detail texturing gives the terrain grit and makes it look much more realistic. Another feature we added is called cubic Bezier curve smoothing. Bezier curves are cubic mathematical equations that give a smooth spline between two points along several control points. The terrain is then adjusted to have its vertices line up with this spline curve, making the terrain look very smooth. This was done to compensate for the fact that taking values straight from height maps can be less than satisfactory, often making the terrain have sharp edges. Because Sphere relies on collision detection, we needed to find an efficient way to allow the main character and all enemies to properly collide with the landscape. In order to do this, every time the main character moves we cycle in new 'chunks' which are collidable blocks spread over the landscape. Only 5 of these blocks will be active at any one time and are the ones directly beneath the main character. Having 5 blocks is much more efficient than attempting to check against all 128^2 blocks every frame, even if the collision culling system is efficient. In order to allow the enemies to collide with the terrain, we implemented a different function 'getHeightc'. This function allowed you to enter in 'X' and 'Z' coordinates and retrieve the exact height of the terrain at that given point. By doing this very efficiently, we were able to reduce the amount of work required to move the characters in the game.

The first sub component of the terrain is the grass. In order to render realistic grass, we started with a good grass texture, and created an alpha channel for it. By doing this, and adding alpha blending to our game, we were able to take a two dimensional picture of grass and make it look much more realistic. We then proceeded to assign certain amounts of grass to each of 32 by 32 blocks over the landscape. When the grass renders, it checks each of the blocks for visibility. If that block is visible, it renders all of its children grass patches; otherwise they are all culled away. This allows us to render much more grass than normal because even though it looks like large quantities, all that is being rendered at any given time is what is directly in front of you.

The other sub component of the terrain class is the 'SphTree' class. This class allows us to easily generate procedural trees and even have them grow during run time. In order to make this realistic for many trees in the level, each tree has its own level of detail system. The trees themselves are made up of cylinders that connect to one another. Each cylinder is capable of having its own level of detail based on your distance from the tree and the

radius of that given branch. By doing this, we are never really overwhelmed by high polygon counts at any given time. In order to make the trees look more realistic, we added the ability to add billboards to the leaves, which make it look much more intricate than it actually is while still running efficiently. To optimize the system even further, we run a culling algorithm before our level of detail system that checks the location of the tree against the viewing window of the camera. If it isn't in the main viewing window, then nothing is calculated or rendered. By culling like this, we are only rendering exactly what is on the screen, which is the most efficient way possible.

In order to unify all these components in a more simplistic manner, we created a small level editor application. In this application, the user is presented with a blank screen where they can load in a height map. They can then take this height map and change any height values of points, blocks or polygons. All changes are updated immediately on screen. Other options include the ability to add grass to the terrain above a certain height level and at a certain density. You can also select sections to mark as places where grass cannot go, such as under puzzles. It also allows you to add trees and change tree properties. When all of your environmental items are placed, you can click the export button, which writes all the code that for what you created into a text file. This code can be copied and pasted right into the Sphere game. By doing it this way, we made it easier to alter the level we're working on and also easier to create or edit any future levels. The following is part of a screenshot meant to show the basic layout of the level editor. See Figure P.

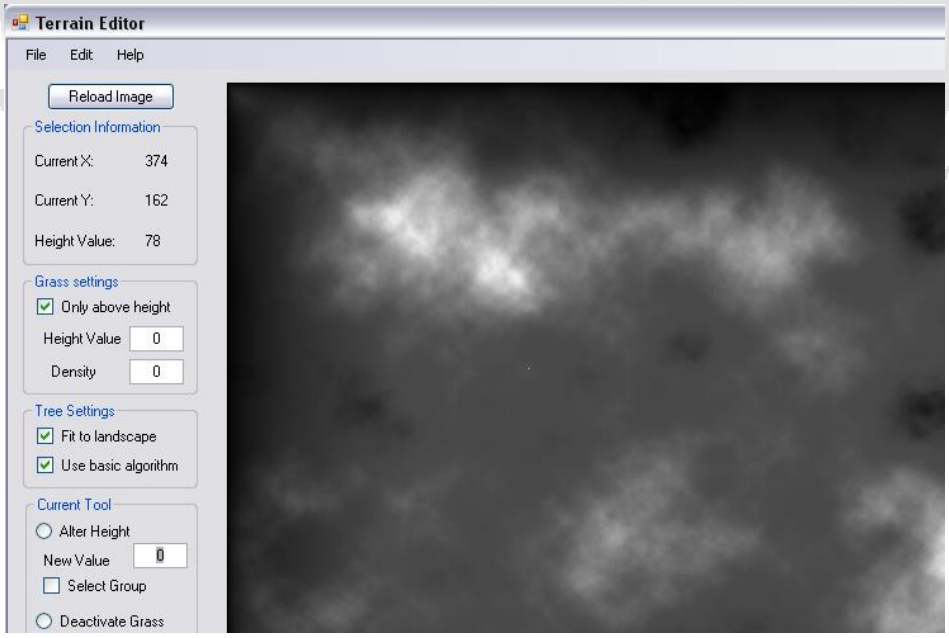

Figure P - (Level Editor Screenshot, Team Sphere)

5.12 Particle Systems

The particle system within Sphere is a fairly simple and standard architecture as far as particle systems go. Each object has a single image associated with it which is instanced for each individual particle. The particles for each particle object is are stored within an array the size of the maximum number of particles. After each iteration, the particle's states are updated sequentially. Various values can be set and changed to influence how the particles are generated and their influence.

Each particle consists of two polygons composing a single GL_QUAD to which a texture is mapped. The quad is then transposed to the calculated location of the particle. A process called "billboarding" is utilized which automatically faces the quad towards the camera.

```
void SphSprayParticles::Render()
{
       int cnt = P.GetGroupCount();
    float *ppos = new float[cnt * 3];
    int num_ret = P.GetParticles(0, cnt, ppos);

        glPushMatrix();
             glBindTexture(GL_TEXTURE_2D,texture);

             for(int n = 0; n < num_ret; n +=3){
                   glPushMatrix();

                   glTranslatef(ppos[n],ppos[n+1],ppos[n+2]);
                   glRotatef(atan2(g_game.m_camera->GetForward()[O_X],
               g_game.m_camera->GetForward()[O_Z]) * 57.29f,
               0.0f, 1.0f, 0.0f);
                   glBegin(GL_QUADS);
                   glTexCoord2f(0.0f,0.0f);
                   glVertex3f(0 - 1.0f, 0 - 1.0f, 0);
                   glTexCoord2f(0.0f,1.0f);
                   glVertex3f(0 - 1.0f, 0 + 1.0f, 0);
                   glTexCoord2f(1.0f,1.0f);
                   glVertex3f(0 + 1.0f, 0 + 1.0f, 0);
                   glTexCoord2f(1.0f,0.0f);
                   glVertex3f(0 + 1.0f, 0 - 1.0f, 0);
                   glEnd();
                   glPopMatrix();

                }
        glPopMatrix();
}
```

Figure Q - (Particle System Render Function Code Sample, Team Sphere)

For each type of particle a separate class was created since each type functioned fundamentally differently.

6 Future Plans

6.1 Distribution

The members of the Sphere Team plan to publicly distribute the Sphere game application for free on their personal websites.

6.2 Content Expansion

At the beginning of the design process, the Sphere team came up with an immense amount of possible features and content to include into the project. If the team were to have more time, they would be able to implement more features and content for the players to utilize within the game.

6.3 Official Website and Forum

The sphere team wished to create an official game website with subsequent forum for discussion of the project. The forum would allow other players to report issues and make suggestions for future features.

6.4 Story + Cutscenes

Robert Lach made the effort to explain the premise of the game within a story. Unfortunately other aspects of the game took priority over implementing the narrative side of the game and the story elements never made it to the game.

In the future we would work on implementing these magnificent tales into the game

Appendices

Appendix A: Game Controls

Keyboard

Move Forward	W
Move Backward	S
Strafe Left	A
Strafe Right	D
Jump	Space
Open/Close Form	Tab
Shoot	Left Control
Regular Closed Form	1
Second Closed Form	2
Third Closed Form	3
Fourth Closed Form	4

Mouse

Turn Camera	Mouse Move
Shoot	Left Click
Switch to Click Mode	Right Click

Appendix B: Installing The Game

1. Download the game from a Sphere Distribution Point
2. Double Click the Installation File
3. Follow Instructions

Appendix C: Starting The Game

1. Double Click on the Executable
2. Follow the on-screen Instructions

Appendix D: License

GNU LESSER GENERAL PUBLIC LICENSE
Version 3, 29 June 2007

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or

b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.

b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.

b) Accompany the Combined Work with a copy of the GNU GPL and this license document.

c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.

d) Do one of the following:

0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.

1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer

system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.

b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

Appendix E: Glossary

Alien – A being not from the planet Earth, highly intelligent and displays far superior technology than that of Earth beings.

C++ - A general purpose programming language.

Earth – Home of the inferior earthlings.

GUI - A type of user interface which allows people to interact with electronic devices such as computers; hand-held devices such as MP3 Players, Portable Media Players or Gaming devices; household appliances and office equipment with images rather than text commands.

OpenAL - A free software cross-platform audio API. It is designed for efficient rendering of multichannel three dimensional positional audio.

OpenGL - A standard specification defining a cross-language, cross-platform API for writing applications that produce 2D and 3D computer graphics.

Polar Bear - A bear native to the Arctic Ocean and its surrounding seas.

SDL - A cross-platform, free and open source software multimedia library written in C that presents a simple interface to various platforms' graphics, sound, and input devices.

Sphere – The best game you'll play in the next 27 minutes, developed by Team Sphere.

Sphere Distribution Point – One of any licensed websites or buildings where Sphere is authorized to be downloaded or distributed.

SuperScrum – A modified iteration of the traditional SCRUM agile software method. SuperScrum takes advantage of faster sprints for shorter development time for projects with a strict deadline.

SuperSprints – A SuperSprint follows the same principles of a sprint in SCRUM, except only allows for a 1 week maximum SuperSprint time.

Team Sphere – The name assigned to the collective developers of Sphere.

User Stories - A software system requirement formulated as one or two sentences in the everyday or business language of the user.