Eat-A-Lot Software

Dude*X

Project Documentation

Version <1.0>

Contributors

Tyler MacWilliam
Ben Randall
Albert Sodyl
Andre Soesilo
Andrew Thompson

Table of Contents

Letter of Transmittal

The following report is a combination of 4 separate and distinct reports developed over the software lifecycle for our game: Dude*X. The reports are:

1) **Game Concept**-outlines the ideas and creative thinking process we went through before designing the game. This is similar to a vision document
2) **Functional Specification**-this document is intended for management and the design team of the game. It gives a more specific overview of all elements of gameplay needed to create Dude*X
3) **Technical Specification**-this document is intended for the game developers and specifically discusses the design of the game based on many of the elements described in the functional specification
4) **Closure Document**-this document is intended for management and anyone who worked on the game. It contains a list of objectives that were met as well as some specific achievements. It also provides some screenshots of the release candidate version of the game. Finally it provides recommendations for future production.

As noted, each of these documents is intended for a separate audience, but as a whole they should be enough to replicate the production of a similar game. They have been compiled here in one document solely for marking purposes for EECE478.

Eat-A-Lot Software

Dude*X

Game Concept

Version <1.0>

Contributors

Tyler MacWilliam
Ben Randall
Albert Sodyl
Andre Soesilo
Andrew Thompson

Table of Contents

List of Figures

1 Introduction

Dude*X is a 3D third person turn-based strategy game for PC, OS X and Linux in which the gamer is challenged to cooperate with "ghosts" of their previous lives to reach the highest floor of the game. The game consists of 5 lives with each life lasting roughly 45 seconds, after which a new "ghost" is created that repeats its actions for the remaining lives. The gamer is required to use these "ghosts" to solve problems and make it up a series of elevators in order to reach the highest floor and win the game.

2 Background

Dude*X is based off of the graphically simplistic, flash-based game Cursor10 (see Figure 2.1) (Ishii, 2008).

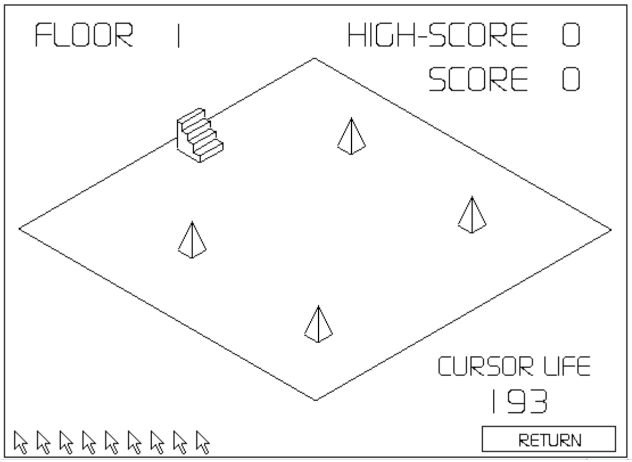


Figure 2.1 Cursor10 Screenshot

In cursor10 the gamer begins with 10 "cursors" that they can use to navigate through 16 floors similar to the one in Figure 2.1. Each time the gamer clicks on the pyramids their points increase. Clicking on the stairs takes the gamer to the next floor. All the actions the current makes are recorded, and when the cursor's life runs out (see bottom right of Figure 2.1) the gamer is reset back to floor 1, and has one less cursor. Nevertheless, all the actions of previous cursors are repeated and shown on screen as "ghost" cursors.

The challenge of the game is that it is necessary to use all 10 cursors cooperatively to make it to the 16th floor. For example, on the 6th floor there is a button that needs to be pressed and held along with 3 other buttons on the 16th floor to win. This means 3 cursor "ghosts" need to spend their entire lifetime getting to the correct floor and holding the button down. Other challenges like this need to be completed, meaning it is very difficult to beat the game the first time around, and even more impossible to collect all the pyramids.

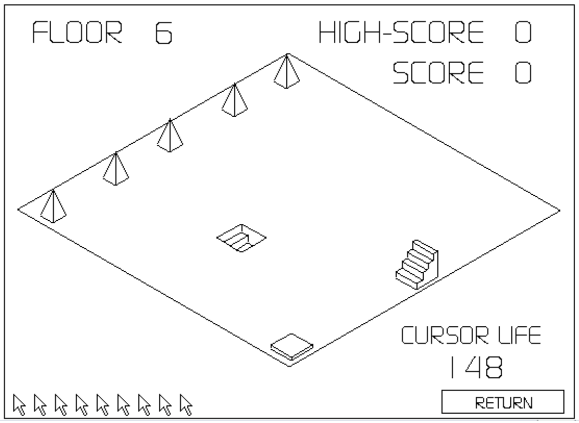


Figure 2.2 Cursor10 6th Floor

Dude10 will be written from scratch by the game developers. For more information about the existing set of code, as well as the various libraries, that will be used in development please see (MacWilliam, 2008).

3 Description

Dude*X will be borrowing some of the game play concepts of Cursor10, but it will be drastically different in its visuals and style. At its core Dude*X will require you to control a "Dude" that will navigate a 3D environment. Like Cursor10, Dude*X will require you to complete a series of tasks on each floor that will necessitate the use of "ghost" dudes (previously run dudes). Each floor will immerse you in its sound and sights and will challenge your ability to see the whole picture. You need to make every movement and click for every Dude count if you have a chance of making it through the all elevators to get to the top floor while collecting all the pyramids along the way. Dude*X is a single player game that feels like a multiplayer game in which the only person you're playing with is yourself.

4 Key features

Dude*X will differentiate itself from Cursor10 and any similar game because of its rich feature set that includes:

- **Portability**: As describe below in section 9 Dude*X will be available for the PC, Mac and Linux.
- **Scripting Language**: it will take advantage of the powerful scripting language Lua to create the levels and sounds. This will allow gamers to potentially develop a customized game for themselves by creating new levels, or even changing the sound; all with only a little bit of

scripting knowledge. By utilizing Lua any changes the gamer makes will not require a recompile, and hence the source code will not have to be released.

- **Transparent and translucent objects:** The "ghosts" will be rendered to be transparent so they can be easily differentiated against the current dude. Some objects (such as glass or water) may be created as translucent objects that will bring an even more realistic feel to the game.
- **Multiple Skyboxes:** A single skybox is used to create the terrain with a secondary moving skybox used to simulate the earth rotating.
- **Level of detail**: Techniques such as mipmapping will be used to maximize game performance.
- **Physical simulation:** The physics engine in Dude*X will simulate real life physics. It will be integrated with collision detection to make the dudes interaction with objects as realistic as possible.
- **Key frame animation**: The Dude will be the most visibly animated object in the game, but the various buttons, switches, etc will also be animated.
- **Collision detection**: As mentioned above collision detection will be an integral part of the physics engine needed to keep Dude*X exciting and fresh. Collision detection will be necessary for the Dude to interact with coins, buttons, switches, balls, etc. that will be present in the game.
- **Shadows**: Shadows will be applied to objects to give them depth.
- **Multi-pass rendering**: Stencil buffer will be used to create reflections on the floor of all objects, including the Dude.
- **Sound effects**: Dude*X's sounds will bring a unique twist to the game. The sounds will all be recorded specifically for Dude*X. Should the gamer not enjoy the sounds they will potentially be able to modify the Lua scripts to use sounds of their liking.
- **User Intelligence**: Unlike most games Dude*X will not require any AI, because the game play will only be as smart as the user playing it. The user is playing with themselves and against the clock; they must solely use their own intelligence to beat the game.

5 Genre

Dude10 is a unique game that is similar to a third person turn-based strategy. However, unlike a turn-based strategy, which is usually multiplayer, the gamer plays every single turn. The gamer's strategy is based on utilizing each turn effectively in order to always have a dude at the right position for the right amount of time. If this does not occur there is no way the game can be beat.

6 Platforms

- Windows 2003/XP/Vista
- Mac OS X 10.4+
- Linux

There is no platform that is preferred. All platforms should play Dude*X correctly. The game speed may change depending on the graphics card, system memory, etc. that the gamer has. Nevertheless, the game should work as designed for the majority of computers that support any of the platforms listed above.

7 Market Analysis

Strategy games continue to be popular with games such as Halo and Everquest continuing to attract gamers. A quick search of strategy game trends showed that Canada and the USA ranked 6 and 7, respectively in their searches for strategy games (Google Inc.). Although Cursor10 is a relatively new game it is trending positively as well. Overall the outlook for this market remains positive.

Nevertheless, even though the strategy game market continues to remain a large market, it has been decided by Eat-A-Lot Software that Dude*X will not be sold for profit. It will either remain solely in the possession of the developers and UBC or may be hosted for free play on the internet by a member of Eat-A-Lot Software.

8 Scheduling

8.1 Risks

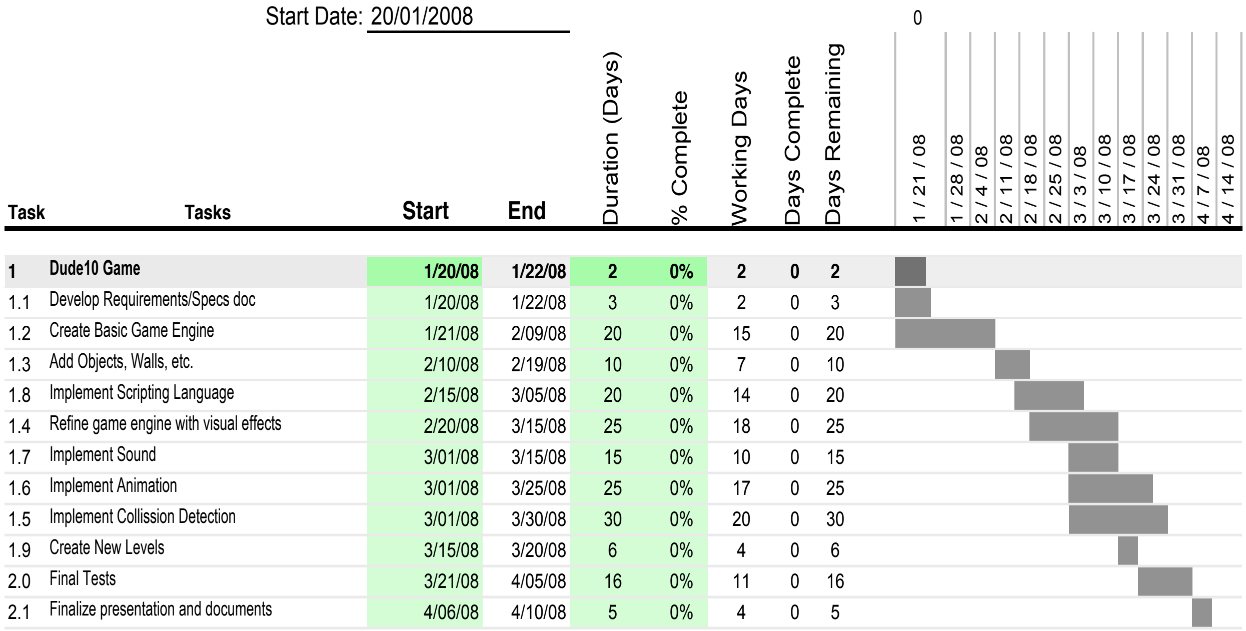
The top 5 risks that could hamper the successful release of Dude*X are as follows:

#	Risk
1	Lack of game development knowledge and experience for team
2	User-interface shortfalls
3	Poor initial design
4	Lack of time to complete all necessary requirements
5	Feature creep

Table 1. Top 5 Risks

8.2 Estimated Schedule

Dude*X is expected to use 5 part time developers to create with a development time of 12 weeks. Figure 8.1 is a representation of the estimated schedule for the development of Dude*X. A majority of the time will be spent creating the basic engine, which includes setting up the cameras, lighting and textures. Once this stage is completed the development team will be able to split off and focus on their areas of expertise. The parallel work structure will culminate with a final product that can be tested by the demonstration date of Apr 18, 2008.



Start Date: 20/01/2008

Task	Tasks	Start	End	Duration (Days)	% Complete	Working Days	Days Complete	Days Remaining
1	**Dude10 Game**	**1/20/08**	**1/22/08**	**2**	**0%**	**2**	**0**	**2**
1.1	Develop Requirements/Specs doc	1/20/08	1/22/08	3	0%	2	0	3
1.2	Create Basic Game Engine	1/21/08	2/09/08	20	0%	15	0	20
1.3	Add Objects, Walls, etc.	2/10/08	2/19/08	10	0%	7	0	10
1.8	Implement Scripting Language	2/15/08	3/05/08	20	0%	14	0	20
1.4	Refine game engine with visual effects	2/20/08	3/15/08	25	0%	18	0	25
1.7	Implement Sound	3/01/08	3/15/08	15	0%	10	0	15
1.6	Implement Animation	3/01/08	3/25/08	25	0%	17	0	25
1.5	Implement Collission Detection	3/01/08	3/30/08	30	0%	20	0	30
1.9	Create New Levels	3/15/08	3/20/08	6	0%	4	0	6
2.0	Final Tests	3/21/08	4/05/08	16	0%	11	0	16
2.1	Finalize presentation and documents	4/06/08	4/10/08	5	0%	4	0	5

Figure 8.1 Gantt Chart

9 Concept Art

This section explores the Dude*X concept in art form. Figure 9.1 is a drawing of the progression of floors (or levels as labeled in the figure) that will make up the game. Currently Dude*X is made up of a title screen and 5 floors. The drawings of each floor are done from a top down view.

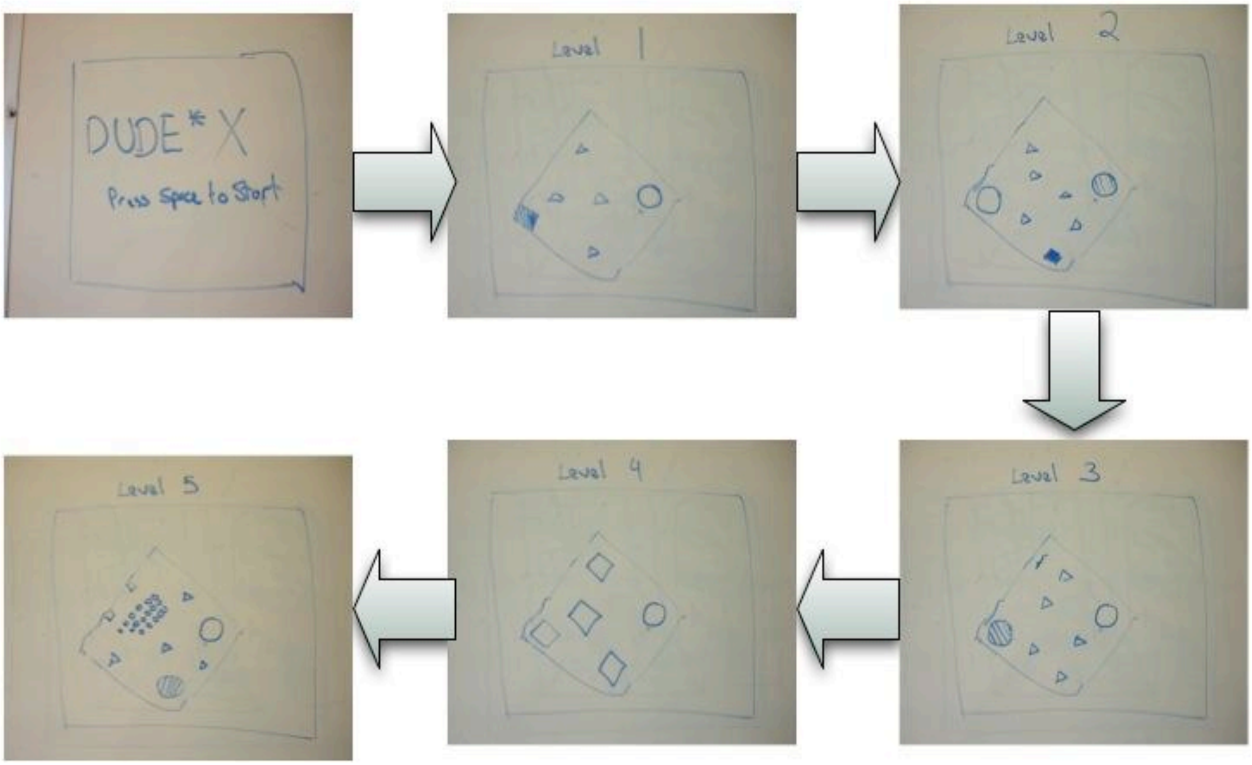


Figure 9.1 Level Concept

Figure 9.2 is a drawing of what the game screen is envisioned to look like at any given point in time. This is the true 3D representation of walls, ceilings, an elevator and multiple pyramids that the gamer must collect to increase their score. There will be a heads up display that notifies the gamer of how many Dudes are left, the time left before the current Dude expires, the current level of the Dude and how many points the gamer has.

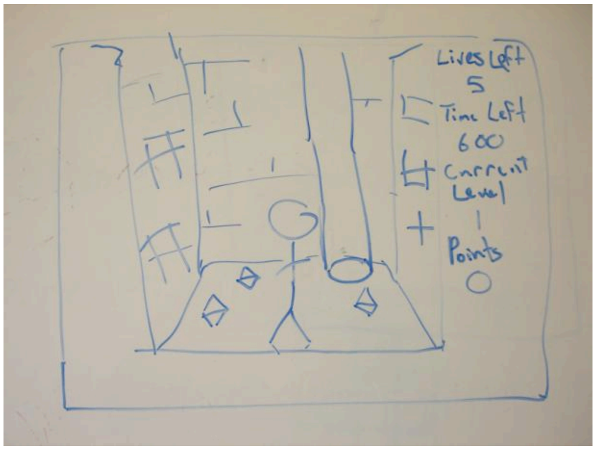


Figure 9.2 Game Screen Outline

Figure 9.3 is a 3D representation of the main game screen (Figure 9.2), but this time it is showing the Dude getting into an elevator. The elevator will transport the Dude to the next floor where there will be new challenges for the gamer to complete as the Dude. The camera will be placed such that it is in the elevator with the Dude when the elevator closes (e.g. comes up from the bottom).

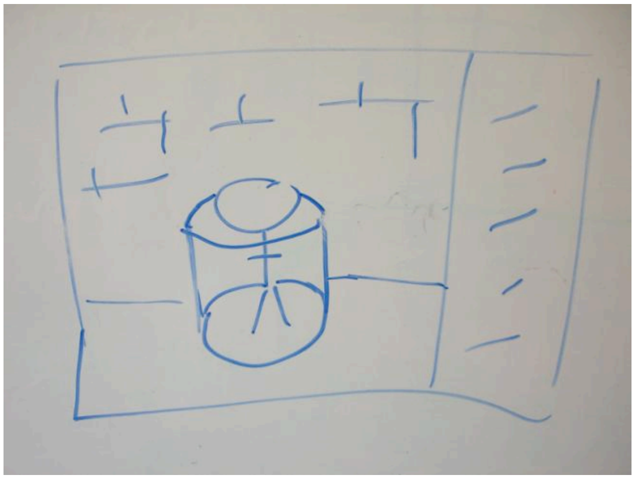


Figure 9.3 Dude getting into elevator

Appendix A: Works Cited

443 Software Inc. ISeeFin Risk Managment Plan. Vancouver, 2008.

Google Inc. Google Trends: strategy game. 23 March 2008. 23 March 2008 <http://www.google.com/trends?q=strategy+game&ctab=0&geo=all&date=all&sort=0>.

Ishii, Yoshio. Cursor*10 . January 2008. 16 January 2008 <http://www.nekogames.jp/mt/2008/01/cursor10.html>.

MacWilliam, T. et al. Dude10-Technical Specification. Vancouver: Eat-A-Lot Software, 2008.

Eat-A-Lot Software

Dude*X

Functional Specification

Version <1.0>

Contributors

Tyler MacWilliam
Ben Randall
Albert Sodyl
Andre Soesilo
Andrew Thompson

Table of Contents

1 Game Mechanics

This section will outline the game mechanics for Dude*X. It will focus on describing the vision of the core game play, followed by the game flow, which traces the player activity in a typical game. It will conclude with sections devoted to characters and game play elements.

1.1 Core Game Play

Dude10 will be a 3D third person turn-based strategy game borrowing some of the core game play concepts of Cursor10 (see section 2.0 of *Game Concept* document), but it will be drastically different in its visuals and style. At its core Dude10 will require the gamer to:

- Control a "Dude" that will navigate a 3D environment.
- Complete a series of tasks on each floor that will necessitate the use of "ghost" dudes (previously run dudes).
- Also collect coins on each level to accumulate their score.

Dude*X is a single player game that acts like a multiplayer game because the gamer is controlling a variety of characters that are cooperating with each other.

1.2 Game Flow

The game flow of Dude*X is quite basic, with the first floor being relatively simple in comparison to the rest of floors, each of which require greater skill to complete the higher the gamer goes. The gamer will be in control of a "Dude" and will use the keyboard and mouse to move the Dude around the level. The levels will each have rotating coins that the gamer can run through to collect points. These will be tallied in a heads up display. Each floor will have one elevator that takes the Dude to the next level, and one level to take the Dude down a level (except for the first level). Once the gamer walks the Dude into the elevator it will close, the screen will fade out, and then fade back in to reveal the next level. The game timer will be stopped during this changing of levels.

Some levels will have buttons that need to be pressed down before the elevator for that floor can be opened up. This will require a Dude to hold this button down, and when the Dude dies its ghost will hold the button down for the currently alive Dude. Other floors will require the Dude to run into large boxes dispersed around the floor in which the elevator to the next level has been hidden. Some floors will require the gamer to run over a button and click their mouse 100 times before the elevator is opened. This task will typically not be possible to complete without the help of a ghost Dude.

The gamer will be under a 45 second time limit for each Dude and they will only have a limited number of Dudes. This information along with the gamers score will be visible to the gamer at all times by way of a heads up display.

1.3 Characters

Dude*X will have a single character: the Dude. The Dude will be created using modeling software that can also create the animations for walking and running. The Dude is a simplistic character in that it does not have any special powers or abilities that need to be tracked. The only characteristic that will need to be tracked is the number of Dudes that are left that aren't ghosts. This number needs to be made readily available to the gamer so they can make an informed decision on how to use all their remaining Dudes.

The Dude will also have some sounds associated with its movement. Please see section 4.1.2 for more information.

1.4 Game Play Elements

This section outlines the variety of elements that the Dude will interact with during the course of a game. Each subsection will provide a description of the element, where the element will appear in the game, and how it will be used by the Dude.

1.4.1 Coins

Coins will be made in real world likeness, but with custom textures for each face of the coin. The coins will be distributed around each floor. They will remain in place and rotating until the Dude or ghost Dude runs over them, at which point they will be "collected" and the gamer's score will be updated accordingly. Each coin that is collected will be removed from view for the duration of the round. They will be reset at the beginning of each new life.

1.4.2 Elevators

Elevators will be made using modeling software, and will provide the method of getting the Dude from one floor to the next. Each floor (with the exception of the first) will have 2 elevators: one the Dude will arrive in and one the Dude will depart for the next floor in. The elevators will be activated by the Dude by stepping on the correct button. On some floors the gamer will also need to complete certain tasks before the elevator can be opened.

1.4.3 Boxes

Boxes will be made in real world likeness, and will be used to hide items such as elevators. These boxes will need to be run into by the Dude, at which point they will be "destroyed" and the item inside (if any) will be revealed for the Dude to collect or use.

1.4.4 Buttons

The buttons will be visible on some levels (see section 6.1) and will require a Dude to press down on them in order for the elevator to open. Some buttons are timed so the elevator begins to close as soon as the Dude gets off the button. These buttons will be placed such that they are nowhere near the elevator. This will make it impossible for a single Dude to use the button and get to the elevator. It will require one or more ghost Dudes to work the button in order for the current Dude to make it to the elevator.

1.5 Game Physics and Statistics

This section will cover the general physics and statistics needed to make Dude*X as realistic as possible for the gamer.

1.5.1 Movement

The Dude should be able to walk along the floor at a quick enough pace that the gamer can get around each floor within 5-10 seconds. Gravity will be present for all elements in the game, meaning elements must be grounded. Coins will technically defy laws of momentum because they will be spinning in place. This is acceptable for the game because it will make the coins more obvious to the gamer.

1.5.2 Collision

Collision detection will be required for all objects that the Dude will interact with. This includes coins, boxes, elevators, buttons, and walls. Each object must have a bounding region that will be used to notify the program when the Dude has entered the bounding region. Depending on what the object is the Dude has collided with certain actions will be performed. If it is a coin, the coin will disappear and the score will be updated. If it is a box it should disappear and reveal its contents (if any). When the Dude collides with a wall it should stop the Dude as a wall would in real life. In levels without walls the Dude should still be stopped on the edge as if it was afraid of heights and does not want to fall off, regardless of what the gamer would like. Buttons and switches should be compressed/flicked when the Dude collides with them. All ghost Dudes must perform the same actions they did when they were alive, including interacting with the same objects (if they are still there).

1.5.3 Statistics

The statistics that must be kept visible to the gamer in order for the game to be successful are:

- Number of lives remaining
- Time remaining
- Score
- Levels completed

However, in order to implement each ghost Dude separate statistics need to be generated based around the movements of the current living Dude, which will later become the movements of a ghost Dude. When the ghost dude appears in the next round its movements will be coordinated by these statistics, such that it can emulate the previous Dude's movements almost perfectly.

2 Gamer Interface

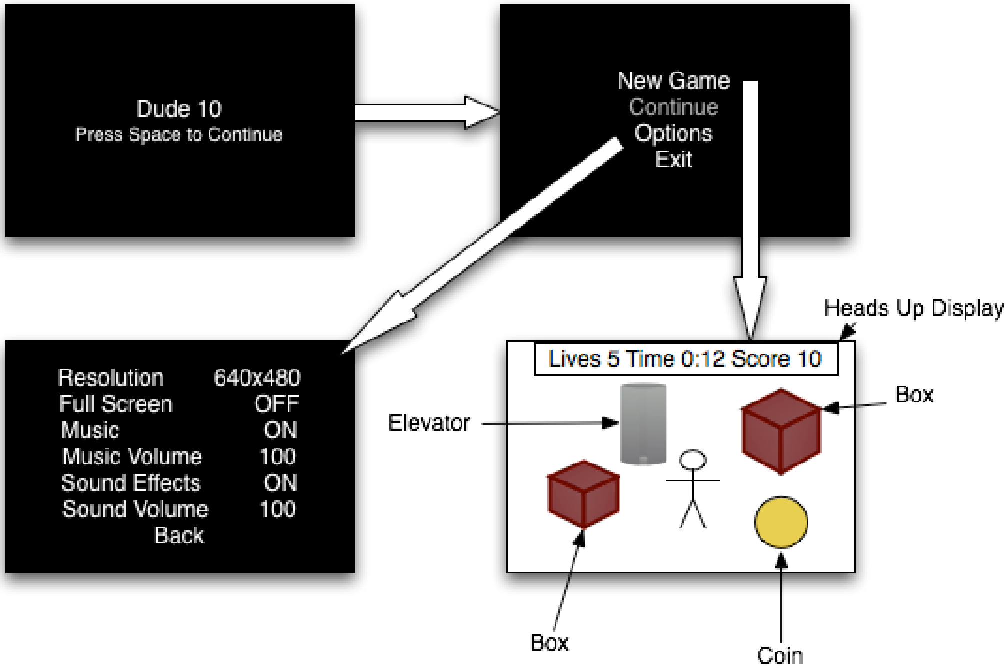
This section outlines the Gamer Interface aspects of Dude*X. It provides a description of the various windows and objects that will make up the Dude*X GUI. It will also provide the functional requirements for some of the key GUI objects.

2.1 Flow Chart

Below is a basic flow chart (Figure 2.1) that outlines the various GUI windows Dude*X will have. There are basically 4 screens:

- The intro screen that appears when the game is first loaded
- The main menu screen that appears after the main screen
- The options screen that can be selected from the main menu screen
- The game screen that appears when Start Game or Continue (if there is a paused game) is selected. The game screen contains a heads up display, the Dude, the floor and walls of the level (not shown) and any game objects for that floor (see section 1.4).

See section 2.2 for a detailed breakdown of each of the windows' functional requirements.



Resolution 640x480
Full Screen OFF
Music ON
Music Volume 100
Sound Effects ON
Sound Volume 100

Figure 2.1 GUI Flowchart

2.2 Functional Requirements

This section will outline the expected outcome for the gamers' actions for each of the screens displayed above in Figure 2.1. Each screen will be discussed in its own subsection.

2.2.1 Main Menu Screen

This screen will notify the gamer that they are about to start playing Dude*X. The expected action is for the gamer to press the Spacebar. This action will take the gamer to the main menu screen (see below). The gamer can also press ESC to quit the game completely.

2.2.2 Selection Menu Screen

This screen will give the gamer four options, which are discussed below. Each option can be selected by using the arrow keys to navigate to the option and pressing the Enter button. This is similar to navigating menus for most computer games and should not differ greatly from the established mental models of the gamer.

New Game

Selecting this option will begin a new game. This entails loading the game screen (see below) with the gamer starting on Floor 1 with the correct initial number of lives, the correct initial time on the clock and a score of 0. All objects must appear in their correct places, such that the first level looks exactly the same every single time a round begins (5 rounds in one game).

Continue

Selecting this option will continue a paused game. It will be similar to New Game, except the game screen will be loaded with the Dude at its last known position and all lives, scores and time reflecting their last known states. All objects that have been collected or destroyed by the Dude prior to pausing the game will remain collected or destroyed.

Options

Selecting this option will take the gamer to the Options screen (see below).

2.2.3 Options Screen

This screen gives the gamer seven options, which are discussed below. Each option can be selected by using the arrow keys to navigate to the option and pressing the Enter button. This is similar to navigating menus for most computer games and should not differ greatly from the established mental models of the gamer.

Resolution

This option allows the gamer to toggle through a variety of screen resolutions that can be used to view the game.

Full Screen

This option toggles on or off the ability to play the game full screen. If it is toggled on the game's resolution will be maximized to fit the size of the gamer's screen. If it is toggled off the screen resolution in the "Resolution" option (see above) will be utilized.

Music

This option toggles on or off the game's background music (see section 4.1.4). If it is set to on the background music will play. If it is set to off the background music will not play.

Music Volume

This is the volume level for the background music. The gamer will use the left and right arrow keys to increase or decrease these numbers. A value of 100 means the max volume and a value of 0 means the background music will be off. A value of 0 will be the same as setting Music to "off" (see above).

Sound Effects

This option toggles on or off the game's sound effects (see section 4.1). If it is set to on the sound effects will play as necessary. If it is set to off the sound effects will not be played.

Sound Volume

This is the volume level for the sound effects (see section 4.1). The gamer will use the left and right arrow keys to increase or decrease these numbers. A value of 100 means the max volume for the sound effects, and a value of 0 means the sound effects music will be off. A value of 0 will be the same as setting Sound Effects to “off” (see above).

3 Art

This section will outline the art needed to give Dude*X its look and feel. It should be noted that Dude*X will be created such that it can be easily customized by the gamer and even Eat-A-Lot Software in the future if need be. The art and video outlined in the subsections will be the basis for what appears in the release candidate version of Dude*X.

3.1 Overall Goals

All art and video created for Dude10 should be based on real world models so they can be recognized by the gamer. However, as long as the model is built on real world models artistic flair can be used to give a custom look and feel. The general art direction should be consistent with a slightly cartoony style, and the mood of the art should evoke a light and possibly happy feeling from the gamer.

3.2 2D Art & Animation

A majority of the art generated for Dude*X will be 2D, because of the strong reliance on texture mapping to create the objects, as well as all the graphics needed to create the menus. This section has been broken down into smaller subsections that outline the 2D art needed to create each of the components.

GUI Screens

The screens have been kept relatively simple (see section 2.1), so the only art that will be needed is to generate the text options for each menu screen. The currently selected option text should be drawn differently such that it is distinguishable from non-selected text. The HUD text should be drawn such that it contrasts against the game screen's background. This may require drawing a filled box behind the text so it can show up more clearly.

Although the mouse will be used during game play a custom icon will not be generated in order to maintain the gamers' mental models. However a custom application icon will be created to distinguish Dude*X from other opened applications.

Terrain

Because Dude*X will require multiple floors, in order to keep the look consistent, only a few textures will be needed. The floors will have a transparent floor, so a tile texture will be used, with each tile transparent. Only the borders between each tile will be seen. A skybox will be developed that will be textured with a cloud scene. This skybox will rotate, as if the earth were rotating. A secondary skybox with a mountain texture will be put in front of the clouds skybox. These two skyboxes will help give the gamer the feeling the level is actually floating in space.

Any floors with walls will have the same brick textures, and the ceilings will also have a brick texture.

Game Play Elements

The coins and boxes (see section 1.4) will be made up of 2D textures for each face. The coin textures will be custom-made for Dude*X, but they will be based off of a Canadian quarter. The coins will be animated by changing their rotation along the y-axis every time the screen is redrawn. The textures for the boxes will be utilizing a wood box texture to give it the appearance of a wooden crate.

Special Effects

Whenever a coin is picked up an explosion of color particles will appear as the coin disappears.

3.3 3D Art & Animation

This section has been broken down into smaller subsections similar to those found in section 3.2. These subsections will outline the 3D art needed to create each of the components critical to Dude*X's successful look.

GUI Screens

There are no 3D components for the GUI screens. Please see section 3.2 for information on generating the 2D art.

Terrain

Although the terrain texture maps will all be using 2D art, the actual instances of terrain and objects must be in 3D. The sky box must be a cube with the sky texture mapped to each face. The walls must be rectangular prisms with the wall texture mapped to each face. The floor must also be a rectangular prism with the floor texture applied to it. It must also have the correct alpha value set such that the skybox may be visible to the gamer and the appearance of a glass floor is maintained.

Game Play Elements

Like the terrain elements the game play elements must be 3D objects with 2D textures mapped to the correct face. The coins must be cylinders with the front and back face textures mapped to the correct face. Boxes must be cubes with the wood textures mapped to each face. All boxes must be the same size.

The elevator will be too complicated to create using the solids API that OpenGL provides, so the elevator model should be developed using modeling software. This software can also create the animation of the elevator opening and closing that can be called when the gamer wants to get in and out of the elevator. Two distinct elevators need to be created: one for going up and one for going down.

The Dude will be too complicated to create using the solids API that OpenGL provides, so the Dude model should be developed using modeling software. This software can also create the animation of the Dude walking and running can be called when the gamer moves the Dude on screen.

Special Effects

Boxes will appear to explode when they are run into by the Dude. This will require animating a box that has a solid state (i.e. not touched) and an animation that has it exploding into pieces (i.e. touched). This will require modeling software. The box will disappear once the animation is complete.

4 Sound and Music

This section will explore the various music and sound effects that will be created and used to take Dude*X to the next level. Like the art in Dude*X (see section 3) the music and sound will be customizable. This will be accomplished through simple modifications of the sound scripts and will not require the gamer to recompile code that makes up the game. However, when shipping the game Dude*X will come with custom built music and sounds that are outlined below.

4.1 Sound Effects

4.1.1 GUI

The GUI itself will not have any sounds. This is because there are not a lot of GUI components and fewer sounds will help to keep the game size relatively small.

4.1.2 Character

The Dude will have a minimal set of voice recordings. These sounds will consist of a “Let’s go!” every time a new Dude starts or a new floor is reached. Every time a Dude dies there will be a moan.

4.1.3 Game Play Elements

Coins will make a “cha-ching” sound when they are collected by the Dude. Boxes will have an explosion sound when broken apart. Switches and buttons will have “click”-esque sounds. The elevator will require a sound for opening and closing.

4.1.4 Environment

Because the game revolves around a closed environment there will not be any ambient sounds, except for the floors which are exposed. This level will have sounds such as airplanes and birds to further give the impression this game takes place in the sky.

4.1.5 Motion

The Dude will have a minimal set of motion sounds, but these sounds will help to make it more lifelike. The sounds will consist of various walking sounds depending on how fast the Dude is moving, and what type of ground the Dude is walking on.

4.2 Music

4.2.1 Event Jingles

There will be a victory song when the game is beat or the gamer runs out of lives and cannot finish the game. Sound effects will be played when the Dude lives and dies (see section 4.1.2).

4.2.2 Shell Screens

There will be a track that plays during the Intro, Main, and Option screens (see section 2.2). Once the game starts the music will change (see below).

4.2.3 Level Theme

Each floor will have its own theme music to clearly define the floors for the gamer. These tracks will be short (<45 secs) and will loop.

5 Story

There is not a back story associated with Dude*X. The game play story consists of getting the Dude to the top floor as quickly as possible. It has been designed to be a simple game that anyone can pick up quickly and be challenged. For more information about the story please see sections 1.1 and 1.2.

6 Level Requirements

This section outlines the design of the levels and how each level will be combined and revealed to make up the game.

6.1 Level Requirements

Dude*X is a linear campaign game in which the gamer begins on the first level and makes their way up to the final level, completing challenges and picking up coins along the way. Below is an outline of the first 7 levels of the game (the expected limit). Please refer to section 9 of the *Game Concept* document for concept art that inspired some of these levels.

6.1.1 Level 1

This is the introductory level. It has been kept fairly simple in order to allow the gamer to get associated with the environment and game play. It will consist of coins and an elevator that will take the gamer to the 2nd level.

6.1.2 Level 2

This level will have more coins, but it will also have a button that needs to be pushed down by the Dude in order to open up the elevator to the 3rd level.

6.1.3 Level 3

This level will be similar to level 2, except this time there are two buttons that need to be held down. One button activates the elevator on the 3rd floor, and the other one activates the elevator on the 5th floor. When a Dude steps off the button the respective elevator for that button will close.

6.1.4 Level 4

This level will consist of a button in the middle of the floor. The Dude needs to step on the button and click 100 times to activate the elevator to the 5th floor.

6.1.5 Level 5

This level will have a single button that needs to be pressed down along with the button on level 3 in order to access the elevator to level 6.

6.1.6 Level 6

This level will consist of a bunch of randomly placed boxes (see section 1.4.3) the gamer must break open in order to find the hidden elevator to the 7th and final floor.

6.1.7 Level 7

This is the final floor. This level will consist of a lot of coins for the gamer to pick up. There will not be a door on the 7th level; the gamer runs around and picks up as many coins as they can before time runs out. Once time runs out a new screen will pop up that will notify the gamer they have beat the game. Any key press will take them back to the main screen.

Eat-A-Lot Software

Dude*X

Technical Specification

Version <1.0>

Contributors

Tyler MacWilliam
Ben Randall
Albert Sodyl
Andre Soesilo
Andrew Thompson

Table of Contents

List of Figures

1 Introduction

This document will be used to define the blueprint design for Dude*X. Each section will outline the design of a specific component of the game. The intention is to combine all sections to create an application that can be tested and will lead up to the final release candidate version of the game.

1.1 Intended Audience

This document is intended to be viewed by the lead programmer, and, as needed, the game developers themselves. The technical language has been kept so as to not lose the details needed to create the envisioned game.

1.2 Game Overview

Dude*X is a 3D third person strategy game for PC, OS X and Linux in which the gamer is challenged to cooperate with "ghosts" of themselves to reach the highest floor of the game. The game consists of 10 rounds with each round of roughly 60 seconds creating a new "ghost" that repeats its action for the remaining rounds. The gamer is required to use these "ghosts" to solve problems and make it up a series of stairs in order to reach the highest floor and win the game.

For more information please see (MacWilliam 2008).

2 Game Mechanics

This section outlines the technical specifications of the systems and code needed to run Dude10.

2.1 Platform

Dude*X will be a downloadable desktop application that works on the following operating systems:

- Windows 2003/XP/Vista
- Mac OS X 10.4.10+
- Linux

The minimum requirements to play Dude*X at the game speed intended the following minimum requirements are required:

- Processor exceeding 1.0 GHz
- 512 MB RAM
- Video card with onboard memory (not required, but will result in better performance)
- Stereo Speakers (not required, but increases game experience)

2.2 External Code

This section will describe the various external code and libraries that will be used to create Dude*X.

2.2.1 SDL

Graphics

Simple DirectMedia Layer is a cross-platform multimedia library designed to provide low level access to audio, keyboard, mouse, joystick, 3D hardware via OpenGL, and 2D video frame buffer (Simple DirectMedial Layer). SDL will be used in Dude*X to handle the user interaction as well as the video management needed to give Dude*X its look and feel.

Audio

To create the unique sound of Dude*X SDL_mixer will be used. SDL_mixer is a special project under the SDL group. It is similar to the standard audio that comes with SDL, except it can handle a variety of audio formats, and can play multiple channels of sound. This means that the game developers can easily program Dude*X to have background music playing while multiple sound effects are be playing all at once, with no effect on the game play . SDL_mixer also provides cross fading and grouping of sounds, enabling the game developers to greatly enhance game play through sound if time permits.

2.2.2 Lua

Lua is a powerful, fast, light-weight, embeddable scripting language. It combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection; making it ideal for configuration, scripting, and rapid prototyping (Pontifical Catholic University of Rio de Janeiro 2008).

Lua will be used by the game developers to easily modify components of the game without recompiling. It will be used extensively to handle scripts of objects and sounds that can be used to dynamically create new levels. Lua can also be used by future gamers to customize components of Dude*X to their liking.

2.2.3 3D Studio Max

3D Studio Max (3Ds) is a full-featured 3D modeling, animation, rendering, and effects solution to produce video and game content (Autodesk n.d.). It will be used in Dude*X to create some of the more complicated objects, as well as the animation (see section 7). 3Ds will allow the developers to create models for the Dude, as well as the buttons and switches needed for the game. These models will also utilize 3Ds' ability to easily develop complex keyframe animation in order to bring more lifelike to the game.

Nevertheless, in order to import 3Ds files into Dude*X a 3Ds loader will need to be created by the developers to read in the 3Ds files and extract the model and animation information. This information will then be used by the game developers and coded into the game through Lua and SDL.

2.3 Control Loop

The main function for Dude*X will set up the correct graphics, video and audio settings for displaying the game. This includes setting the screen size and enabling the various OpenGL properties that are require for the game (such as the shading mode and culling settings). Some of these settings will be based on the gamers' preferences that are saved each time the game is played.

Once the correct graphics, video and sound have been set the peripherals are initialized. This includes setting up the managers for the keyboard, mouse, screen and any other events needed during the game. Finally the Lua scripting engine will be initialized. This sets up the basic Lua functionality that is required by the majority of the game components. No, game specific content is loaded at this time. Specific levels level information (walls, models, sounds, etc.) are loaded when they are needed. Once the Lua engine is complete the main menu is displayed and the main event loop is entered.

The main event loop will handle the timer that will synchronize all the updates. It will handle calling the correct update routines utilized by SDL and Dude*X. Because each major component of the game (screen, keyboard, mouse, Dude, coin, elevator, sound, event manager, etc) will have their very own update methods, the main event loop will handle calling them at the appropriate time. Once all the update methods have been called the updated screen will be displayed for the gamer. The goal is to achieve a frame rate of greater than 60 frames per second.

For a detailed view of the flow of the main game control loop please see Figure 2.1 below. A detailed breakdown of the event loop within the main game loop can be found in Figure 2.2.

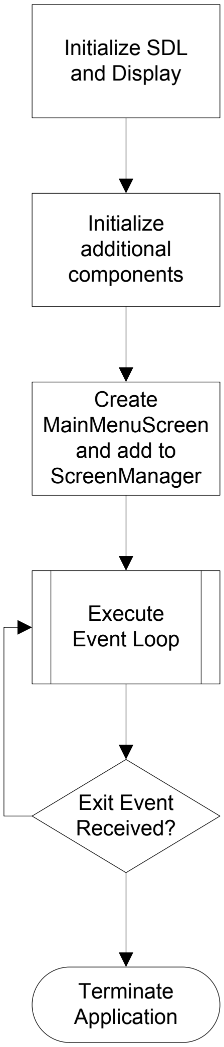
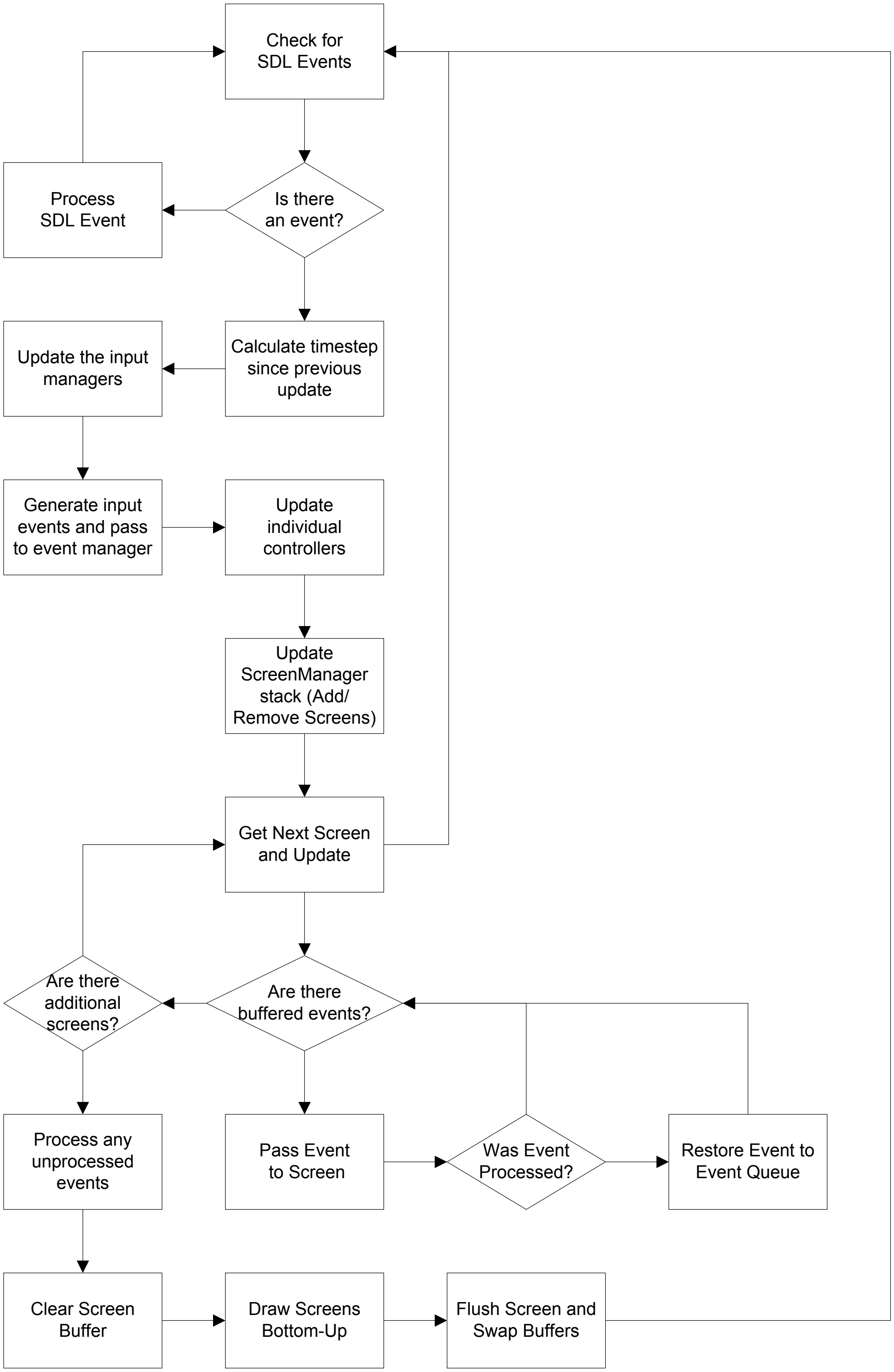

Initialize SDL and Display
Initialize additional components
Create MainMenuScreen and add to ScreenManager
Execute Event Loop
Exit Event Received?
Terminate Application

Figure 2.1 Game Loop Flowchart



Check for SDL Events
Is there an event?
Process SDL Event
Calculate timestep since previous update
Update the input managers
Generate input events and pass to event manager
Update individual controllers
Update ScreenManager stack (Add/ Remove Screens)
Get Next Screen and Update
Are there buffered events?
Are there additional screens?
Process any unprocessed events
Pass Event to Screen
Was Event Processed?
Restore Event to Event Queue
Clear Screen Buffer
Draw Screens Bottom-Up
Flush Screen and Swap Buffers

Figure 2.2 Event Loop Flowchart

3 Game Object Data

Because Dude*X will require the managing of a plethora of objects, instead of discussing each object in detail the following class diagram has been given as an outline for the design and interaction of all objects within the game. Discussion of major classes will be discussed after the figure.

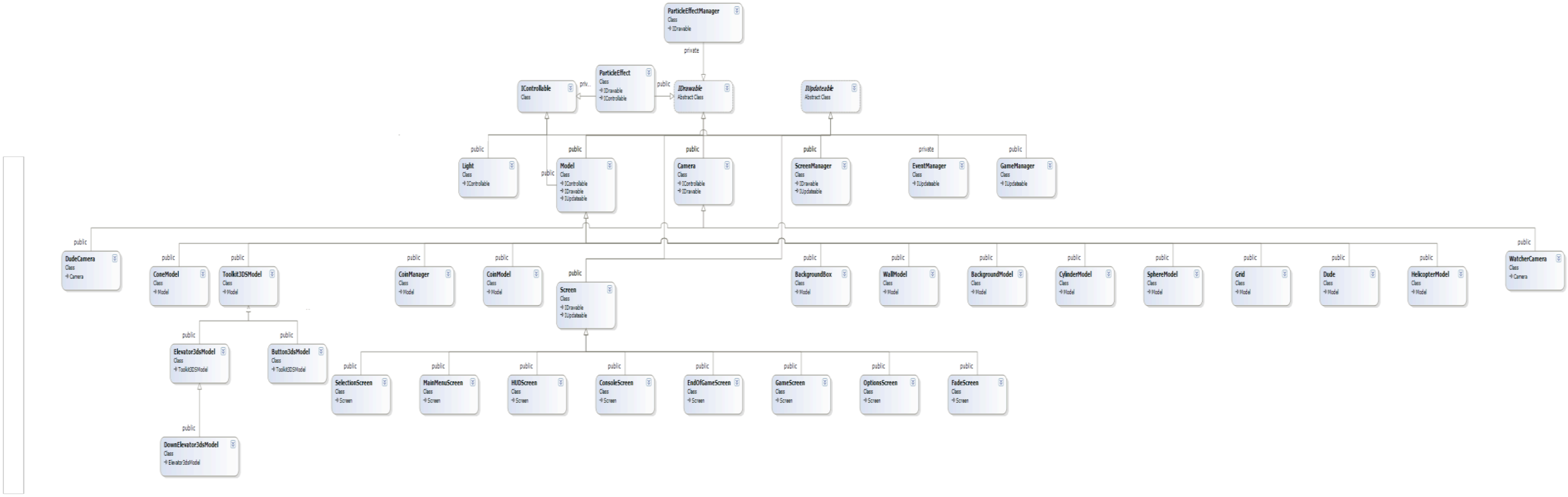


Figure 3.1 Main Class Diagram

3.1.1 Screen

Screen is an abstract class that implements the IDrawable and IUpdateable interfaces. It defines key virtual methods draw() and update() that must be defined by all classes implementing Screen. These classes include:

- SelectionScreen
- MainMenuScreen
- HUDScreen
- ConsoleScreen
- EndOfGameScreen
- GameScreen
- OptionsScreen
- FadeScreen

Each of these screens will be accessed and drawn to the screen as needed through a ScreenManager class, which will contain a list of all screens, as well as a list of buffered screens. For a detailed breakdown of what each screen is for please see section 6.

3.1.2 Event

Figure 3.2 outlines the implementation of the abstract Event class that is used to define events that are handled in the game. The virtual methods that need to be implemented in each class that inherits Event are play(), getType(), getModel() and getTarget().

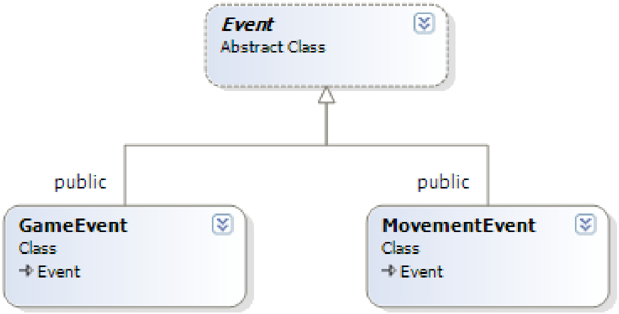


Figure 3.2 Game Event Class Diagram

3.1.3 Controller

Figure 3.3 outlines the implementation of the abstract IController class that is used to define the various controllers in the game. All controller classes implement the abstract IController class. The virtual methods that need to be implemented in each class that inherits IController are attachTo(), update(), isEnabled(),and setEnabled(). All controllers are managed by the ControllerManager class that has an array of IController objects. The ControllerManager class allows an easy way to add, remove and get the various controllers active in the game.

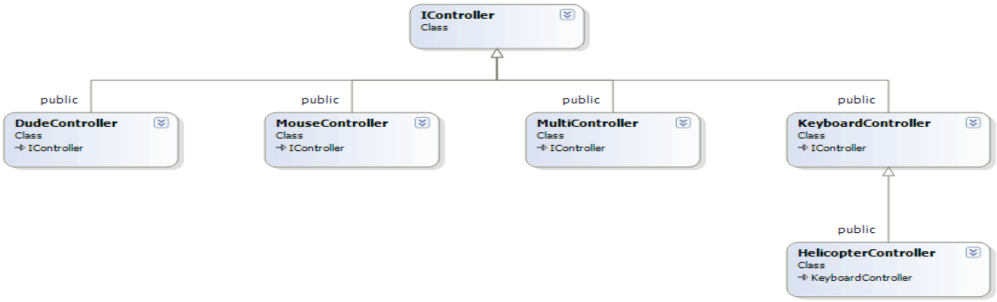


Figure 3.3 Controller Class Diagram

3.1.4 Lua

Figure 3.4 outlines the implementation of the abstract ILuaFunctExporter class that is used to define the interface for all classes wanting to interact with the Lua scripting engine. The Lua scripting engine is handled by the LuaC class and Lua functions can be called in game through the game console (Console class) or through Lua scripts. The ILuaFunctExporter defines two static functions:

- registerFunct(lua_State *pLuaState)-exports function to a Lua State
- registerFunct(LuaC* luaObj) – exports function to a LuaC object

Every class that inherits ILuaFunctExporter must create an array of LuaFunctions (defined in LuaC) that will be called from Lua scripts or through the console. These LuaFunctions are pointers to functions defined in the inheriting class (i.e. LuaObjMgr). Once the array of LuaFunctions is initialized with the correct function pointers the class must also implement both registerFunct functions, which will export the functions into the correct Lua states and LuaC objects. Once the LuaFunctions have been registered correctly they can be called from within the Console or through a Lua script.

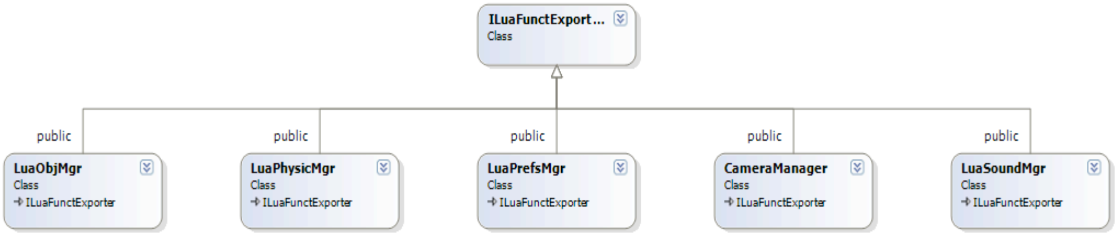


Figure 3.4 Lua Class Diagram

Lua will be used for testing and making large changes to the game without the need to recompile. As such, some critical components that will need to have Lua implementations are:

- Objects (walls, floors, coins, etc)
- Physics (bounding boxes)
- Preferences (resolution, fullscreen)
- Sound (sound effects, background music)

Each of these components will be initialized and controlled through a component manager class (i.e. SoundManager) that will have functions to access and use a list of loaded components (i.e. play or pause music). Together these Lua components will be used to create Lua scripts of each level defined in section 6.1 of the *Functional Specification* document (MacWilliam 2008).

3.1.5 3DS Models

Figure 3.5 outlines the implementation of the 3DS models in the game. Toolkit3DSModel inherits from the abstract Model class. Toolkit3DSModel implements some critical functions that are needed for animating any object. These functions include:

- draw(void) –transforms and draws the 3Ds mesh
- update() – updates all meshes
- StartAnimation(int animationNumber) – starts the animation
- StopAnimation(int animationNumber) – stops the animation
- ResetAnimation(int animationNumber) – resets the animation
- void setPlaybackSpeed(float value) – sets hows fast or slow the animation runs
- setLooping(bool value) – sets whether or not the animation should loop

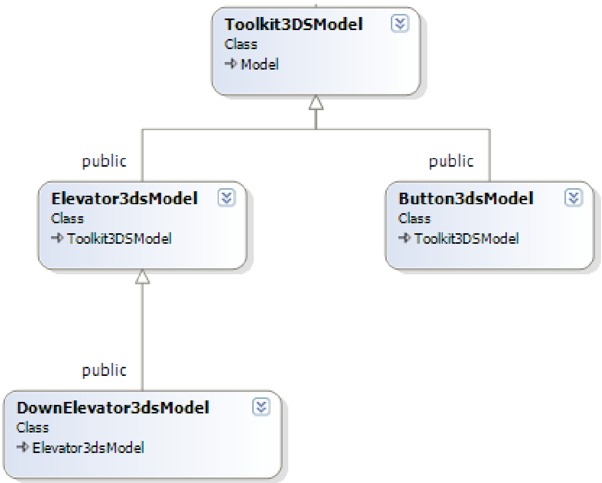


Figure 3.5 Toolkit3DS Class Diagram

With the exception of loading the Dude model (which creates a Toolkit3DSModel of the Dude within the Dude class) the only other 3DS models implemented in the game are the up elevator (red teacup), down elevator (blue teacup) and the buttons that the Dude will stand on. These each inherit and implement functions from Toolkit3DSToolkit so they can work specific to their model (i.e. the up elevator runs an animation that opens and closes the elevator door), see Figure 3.1.

4 Game Physics

As part of the Dude*X game engine, the physics engine will provide support for modeling physical objects in the game world. Essentially, it will be responsible for controlling the movement of objects and their interaction, i.e. collision detection and response.

4.1 Physical Objects

Three kinds of physical objects will be possible in the engine:

- Dynamic objects –objects that move between frames and have associated:
  - Position
  - Rotation
  - Velocity
  - Mass
  - Angular velocity
  - Friction coefficient of the bottom
  - Elasticity
  - Any forces that are applied to it
  - Bounding volume
- Static objects –objects that do not move between frames and thus these objects consist only of a bounding volume.
- Space bound –the bounding volume of the game world. Although it is similar to a static object, it will be treated slightly different.

4.2 Bounding Volumes

Bounding volumes are closed shapes that contain the physical objects of interest and are geometrically simpler than the objects to improve the efficiency of geometric operations. Currently, the physics engine will support only a single bounding volume per object; in the future it could be expanded to follow the models and have possible bounding volumes at each tree node. The engine will support three kinds of bounding volumes:

- Sphere
- Cuboid (a.k.a. Box)
- Cylinder

For each volume there will be an option of whether collisions should be monitored inside the volume or outside. This is in part a performance optimization that reduces the number of bounding tests necessary. The "inside" variant could be used for the box volume shape to keep elements inside the box, which is useful for the space bound. Usually, the "outside" variant will be used since not many objects will be encompassed within other objects in Dude*X.
The cuboid volume shape is the most complex shape because it tests against six finite planes. Since collision detection and response for non-axis aligned finite planes is complex and not necessary for our game, the engine will only support axis-aligned bounding boxes. However, for the "inside" case, infinite planes can be used, and thus support non-axis aligned boxes for that variant. This means that the space bound does not have to be axis-aligned.

Cylinders are infinite in length. Although collision detection for finite cylinders is not difficult, calculating the response would be more involved and is not necessary for the game. Sphere bounding volumes are the simplest and specified by their radius. A radius of zero would create a point that could be used for some collision detection scenarios.

4.3 Collision Detection and Response

Collision detection will work by finding the distance between bounding volumes at the next frame. If a collision occurs at the next frame, then the collision response is calculated and the physical object is diverted from the collision. This process occurs only to dynamic objects, as static objects do not move between frames. For each dynamic object, the following will occur for each frame:

1. Any current forces and acceleration, including gravity are applied.
2. Velocity from acceleration is updated
3. Any constant velocity setting from controllers is applied
4. Test against collisions between other dynamic objects
5. Test against collisions between static objects
6. Test against collisions between the object and the space bound
7. Apply any collision response if there was a collision
8. Actually move the object by translating it by its velocity
9. Update and apply friction

Note that all dynamic objects are tested against all other objects, meaning a complexity of $O(n^2)$ is used for collision detection, where n is the number of dynamic objects. This solution will be acceptable since there will not be many dynamic objects, and collision detection functions should actually take a negligible amount of execution time compared to the rest of the game.

Since collisions will involve different combinations of static and dynamic objects as well as different shapes, only collisions that are needed will be implemented, namely the following is a list of collisions will be supported by the physics engine:

- Sphere vs. static cuboid
- Sphere vs. static infinite cylinder
- Sphere vs. static sphere

- Sphere vs. dynamic sphere
- Infinite cylinder vs. static cuboid
- Infinite cylinder vs. static sphere
- Infinite cylinder vs. dynamic sphere

Details of the math for each collision will not be discussed here. Collision detection for each combination of shapes is a bit different, but collision response will almost always uses the same formula:

$$\vec{R} = 2(-\vec{I} \bullet \vec{N})\vec{N} + \vec{I}$$

Where ***R*** is the response vector, ***I*** is the incident vector, and ***N*** is the vector normal to the surface of the collision. In each case, the methods to determine the normal will be slightly different and sometimes approximations are used. The incident vector will usually be the velocity.

.

5 Artificial Intelligence

As mentioned in the *Game Concept* document (MacWilliam 2008) “Dude*X does not have any AI”. The gamer must use all the Dudes they have and their own intelligence to complete the game. This section discusses how the “artificial intelligence” of the “ghosts” will be developed to accurately re-enact, step by step, the previous dudes’ actions.

In order to record what happened in a previous round, there are two approaches that can be taken. The events of interest may be recorded, or the gamer input could be recorded. With either method, the recorded entities would be played back for each round. Event-based recording (see section 3.1.2) should be used because it is more efficient and would rule out any accuracy issues that might be faced if going with input-based recording instead. This type of recording is different than a “timedemo” because interaction still occurs with the recorded state of the game.

There are two kinds of events that will be recorded: continuously sampled events that are “polled” and action-driven events that are “pushed”. These will be called MovementEvents and GameEvents, respectively. Other classes or subclasses of these events may be created to specify alternate event conditions. Both will be subclasses of an abstract Event class that contains the time the event occurred, and an interface for how it should be replayed. MovementEvents will be recorded at a specified sampling frequency of around 60 Hz for smoothness, and will contain the position and rotation of the recorded character. GameEvents contain information on the type of event, such as a coin pickup or button press, and the associated game model with that event. All events will be stored in a linked list that is traversed as the events are played back, and any new events will be inserted as they occur.

6 Gamer Interface

This section outlines the design of the gamer interface necessary to correctly play Dude*X. The Game Shell section will explore the design of all screens other than the Main Play Screen, which will itself be outlined in section 6.2. The intention is that this section will provide enough detail for the lead programmer and game developers to understand and implement Dude*X's gamer interface.

6.1 Game Shell

As mentioned in section 3.1.1 there are a variety of screens that will be handled by the ScreenManager class. Each screen implementation is discussed in their own section below.

6.1.1 MainMenuScreen

The MainMenuScreen class will implement the Main Menu Screen discussed in section 2.2.1 of the *Functional Specification* (MacWilliam 2008). It will be the first screen loaded and displayed when the gamer starts up Dude*X. The screen should be simple and should display the name of the game and any instructions needed to take the gamer to the next screen: the SelectionScreen. When the next screen is loaded it should be pushed on to the list in ScreenManager. MainMenuScreen will remain in the list until the game is exited.

6.1.2 SelectionScreen

The SelectionScreen class will implement the Selection Menu Screen discussed in section 2.2.2 of the *Functional Specification* (MacWilliam 2008). It will have to draw each of the gamer-selectable menu items, which can be highlighted using the up or down arrow keys. The SelectionScreen class will also need to be able to process which menu option the gamer has selected. If the New Game option is selected a new game will need to be loaded. This means fading the screen (see section 6.1.7) and loading the game and HUD screens (see sections 6.1.3 and 6.1.6). If a game is in process the Continue option can be selected which will return the gamer back into the game at the same position they paused the game by popping the SelectionScreen off the ScreenManager stack. Selecting the options screen will push the OptionsScreen onto the ScreenManager stack which will draw the OptionsScreen. Selecting the Exit option will push an end game notifier to the event queue, which will then properly dispose of all objects correctly and shut the game down.

6.1.3 HUDScreen

This class will handle drawing the heads up display that will be visible to the gamer at all times. The HUD will show the gamer the number of lives remaining, the level they are on, the points they have achieved and the time remaining. These values will be retrieved from the GamerManager class and will be updated every time GameManager's updated method is called and the screen is redrawn. It is not interactive, thus the only event that needs to be handled is the end game event. In this case the HUD screen will be popped from the list in ScreenManager so it is no longer displayed.

6.1.4 ConsoleScreen

This class will handle drawing the drop down console screen defined in the Console class that can be used to interface with Lua (see section 3.1.4). The instance of the Console class will be accessed by the ConsoleScreen through the Singleton design pattern. The Console class' draw method will draw a background box for viewing commands over top of the GameScreen and HUDScreen, and a separate background box for entering commands. The gamer can type Lua commands and when they press enter the input will be processed by Lua and the command along with any output will be drawn to the background box for entering commands. When the Console is deactivated the ConsoleScreen will be popped out of the list in ScreenManager, so it will no longer be drawn.

6.1.5 EndOfGameScreen

This class will handle the drawing of a simple screen on top of the GameScreen that will help to notify the gamer they have lost. A message telling the gamer they have lost is appropriate. When the game is exited or the gamer wants to return to the SelectionScreen this screen is popped and an EndGame event is pushed onto the InputEvent queue.

6.1.6 OptionsScreen

This class will handle the drawing of the Options screen discussed in section 2.2.3 of the *Functional Specification* (MacWilliam 2008). It will have to draw each of the gamer-selectable option menu items, which can be highlighted using the up or down arrow keys. The OptionsScreen class will also need to be able to process which option the gamer has pressed the left or right arrows because these will be used to change the values of the currently selected option.

For the resolution option there will be a list of accepted game resolutions that the gamer can select from. When pressing the left arrow the next resolution needs to be retrieved and displayed. If the gamer presses the left arrow the previous resolution needs to be retrieved and displayed. Each time the resolution is changed the resolution is written to the preferences file defined in the Preferences class.

The gamer can select between fullscreen on and off. Like resolution, each time the option is changed it is written to the preferences file.

The background music can also be toggled on or off. Each change will be written to the preferences file.

The music volume can also be set. Pressing the left arrow key will decrease the volume by 1, and pressing the right value will increase it by 1. 100 is the maximum volume level and 0 is the minimum.

Sound effects can also be toggled on or off. Each change will be written to the preferences file.

The sound effects volume can also be set. Pressing the left arrow key will decrease the volume by 1, and pressing the right value will increase it by 1. 100 is the maximum volume level and 0 is the minimum.

Selecting the Back option will save all the preferences and pop the OptionsScreen off the list in ScreenManager, thus revealing the SelectionScreen again.

6.1.7 FadeScreen

This class will handle the drawing of a screen that will fade in and out for a developer programmed period of time (most likely 1-2 seconds). The FadeScreen will be pushed on top of all screens in the ScreenManager list when a new level is loaded. The fade colour should be set as white. Alpha blending will be used for both fading in and out, thus GL_SRC_ALPHA and GL_ONE_MINUS_SRC_ALPHA should be used for the glBlendFunc. When fading in the Alpha value should be set to 0 and should increase linearly in steps of

$$A = \frac{fade\ time}{fade\ duration}$$

where A is the new alpha value. When the fade is complete the Alpha value needs to be set to 1, just in case the fraction is not exactly equal to 1.

For fading out, which will occur once the new level has loaded the alpha should start at 1 and the alpha fade out value should be

$$A_{fadeout} = 1 - \frac{fade\ time}{fade\ duration}$$

6.2 Main Play Screen

The main play screen is the most complicated of all the screens previously discussed. Whenever the main play screen is loaded (through a call to its load function) the cameras, models, lights and controllers all need to be initialized (see section 7.1). Each of these components will have their own initialization functions to make the code easier to understand and use. Once all the components have been initialized the Dude model can be activated and all mouse events will be cleared. This is done so the camera (which is controlled by the mouse) can always start by facing the same direction every time the game screen is loaded.

Drawing the main play screen is complicated because each component needs to be drawn in the correct order. First the background skybox is drawn, followed by the terrain skybox. The lighting effects are then implemented by calling the draw function of the LightManager class. Once the lighting has been set all objects (Dude, coins, walls, etc.) are drawn by calling the draw function of the DrawManager class. Finally the cameras and particle effects are drawn.

Every time the main play screen's update method is called the background is updated, any events that occurred are updated and recorded, and finally all objects (Dude, coins, walls, etc.) are updated. Once all the correct update methods are called the main play screen is redrawn to reflect the updates.

7 Art and Video

This section discusses the details of the graphics engine and the creation of any animations. The graphics engine will discuss the implementation of cameras, textures, models lighting and fonts that will be needed to give Dude*X it distinct look.

7.1 Graphics Engine

The graphics engine of Dude*X is responsible for translating the effects of the rest of the game engine into images which are rendered to the screen. This consists of a number of parts including Cameras, Models, and Lighting, among other smaller components.

7.1.1 Cameras

The basic purpose of the camera object is to provide a view point and orientation through which the screen is rendered. The camera can represent a variety of things. A free camera can provide an overview of the map/game. A fixed camera representing an actual camera object in-game through which the character can view some other part of the world. A controllable camera can represent the characters view, either in first person or third person depending on the method used to control the camera.

As the game will require a number of these different camera types a camera system needs to be developed to allow multiple cameras to exist at any given time. From this point, any given camera can be activated, at which point the scene being rendered will appear from the viewpoint of the active camera. The camera objects should be controllable in the same manner as other game objects to allow ease of control.

7.1.2 Textures

Textures are often a very resources intensive part of a game and as such, there needs to be a method to manage textures within the game. A texture manager should keep track of all textures that are loaded and ensure that any given texture is only loaded once. Additionally, the texture manager is responsible for generating mipmaps for the loaded textures to improve game performance.

7.1.3 Models

A model represents the abstract concept of any object that can be draw within the game. To simplify management of objects, all drawable objects should inherit from the same base class, IDrawable. IDrawable provides a method draw() which is overridden to provide the actual drawing functionality of the model. This functionality will vary depending on the model.

Simple models such as the Skybox will draw a number of quads textured with the skybox. This involves specifying the textures that will be used for each side of the skybox, and drawing a single textured quad for each section of the box. The skybox object is always positioned in the same place as the current camera to produce the illusion of a distant background.

More complex models such as the game walls are not constantly the same shape and thus are not well suited for generating as 3DS models. Walls must also be affected by game lighting so they must be made up of larger numbers of quads. This is done by subdividing the wall into an arbitrary number of chucks and drawing a quad for each. The quads are textured so as to seamlessly repeat the texture. The display lists for the walls should be generated a single time to maximize performance.

Finally, more complex in game models (3DS models) are loaded into the game using a 3DS file parser. This includes a number of steps. First, the meshes are loaded into an object. As 3DS files do not contain normal vectors for vertices, these are now generated. Second, all animations are loaded from the file and converted into the internal format. Finally a display list is generated for each mesh. Each mesh has an individual display list to allow run-time modification (animation) of individual meshes in a model. During this step, any materials or textures required are created or loaded.

During the drawing step for any model, any specific transforms are applied which positions the model in the scene with respect to the camera. Second, the drawing code is executed. This usually includes a display list but can include other OpenGL calls to manually create objects. For a 3DS model this involves transforming each individual mesh before drawing it.

7.1.4 Lighting

Lighting is an important part of any 3D game as it provides a way to make objects appear much more 3D and realistic. There must be an easy way to interact with the lighting system to allow creating new lights, and enabling and disabling lights. Similar to most other components, lights must be controllable to allow them to be animated, positioned, and controlled easily.

7.1.5 Fonts

Text is the standard way to provide feedback to the gamer whether it is in the form of a menu system or an in-game heads up display. The graphics engine should provide a way to load a number of pre-generated font textures into the game and then use them to print various strings to the screen. Similar to the texture manager, fonts should only be loaded a single time and re-used by the various components of the game.

7.2 Artist Instructions

A 3Ds models will need to be created of the Dude. The Dude should be simplistic and gender-neutral. It must be simplistic such that any animations will be created (i.e. walking/running) will not be difficult to parse and implement in the game. Any elevators, buttons and switches should also be completed in 3Ds to give them the detail and accurate animation.

Textures will need to be created for the walls, floor, balls, coins and boxes. Any textures created for the game must have dimensions equal to 2^n. All textures should be accurately cropped in production such that alpha blending will not need to be utilized except in necessary conditions.

8 Sound and Music

This section outlines the implementation of sound in Dude*X. It begins with a brief discussion of the various requirements for recording background music and sound effects in order for them to play properly in the game, and concludes with a brief overview of how the code will need to be developed in order for the correct sounds and music to play at the appropriate times.

8.1 Sound Engineering

Background music can be encoded in any currently accepted music format. However, in order to keep the size of the background music minimal it should be recorded in MP3 or AAC format at around 128-192kbps. Background music can be longer, but should be able to loop without the gamer noticing. Looping is inevitable if the gamers spends their entire time on the same floor; unlikely, but still possible.

Any sound effects need to be recorded in WAV format and must be dual channel. The frequency should be set at 22050 kHz and should be encoded using 16 bit waveforms. Sound effects should be no longer than 5 seconds.

8.2 Level Specific Code

All code will be handled by a Sound Manager that will be created by the game developers. The functions of the Sound Manager will include those that can load, play and pause both background music and sound effects. These functions should be accessible through the Lua scripting interface, such that all sound effects can be loaded through a single file at the start of the game through the Lua initializing function (see section3.1.4). Code that calls the correct sound effect ID to play for a specific event (i.e. coin is collected) will need to be implemented by the game developers at the correct spot in code (i.e. collision with coin is detected).

The background music will also be loaded at the start of the game, but because it is level specific it will be handled in each level's Lua file. These level files will be read in the Lua initializing function (see section 2.3) and the background music will be loaded at the same time. Code will need to be developed such that when a new floor is loaded the correct background music plays as the floor appears (i.e. in the Level Manager). The background music will help to differentiate the various floors.

Appendix A: Works Cited

Autodesk. *Autodesk-Autodesk 3Ds Max.* http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=5659302 (accessed January 31, 2008).

Gold Standard Group. *GLUT-Open GL Utility Toolkit.* http://www.opengl.org/resources/libraries/glut/ (accessed January 15, 2008).

Ishii, Yoshio. *Cursor*10* . January 2008. http://www.nekogames.jp/mt/2008/01/cursor10.html (accessed January 16, 2008).

MacWilliam, T. et al. *Dude10 Game Concept.* Vancouver: Eat-A-Lot Software, 2008.

Pontifical Catholic University of Rio de Janeiro. *Lua:About.* January 7, 2008. http://www.lua.org/about.html (accessed January 25, 2008).

Simple DirectMedial Layer. http://www.libsdl.org/index.php (accessed January 15, 2008).

Eat-A-Lot Software

Dude*X

Closure Document

Version <1.0>

Contributors

Tyler MacWilliam
Ben Randall
Albert Sodyl
Andre Soesilo
Andrew Thompson

Table of Contents

List of Tables

List of Figures

1 Introduction

This document has been produced to outline the achievements, performance and future recommendations for Dude*X. It is intended that the recommendations can be used when developing future and similar games as that of Dude*X. It is intended for management as well as anyone who participated in the development of the game.

2 Achievement

The Dude*X project originally began with a simple concept in mind: a 3D third person strategy game for PC, OS X and Linux in which the gamer is challenged to cooperate with "ghosts" of them to reach the highest floor of the game (MacWilliam, Dude10 Game Concept 2008). This concept was further refined through the creation of 2 critical documents: the Game Concept (MacWilliam, Dude10 Game Concept 2008) and Technical Specification (MacWilliam, Dude10-Technical Specification 2008). Although the requirements changed over time the fundamental concept remained the same.

This section gives an overview of the objectives of the project as well as some of the key milestones reached during the creation of Dude*X. It concludes with a brief section on the final results of the Dude*X game.

2.1 Original Objectives

Of the features outlined in both the Game Concept (MacWilliam, Dude10 Game Concept 2008) and Technical Specification (MacWilliam, Dude10-Technical Specification 2008) the following were met:

Objective	Status
Portability	Achieved
Shadows	Missed
Scripting Language	Achieved
Transparent and translucent objects	Achieved
Multiple Skyboxes	Achieved
Level of detail	Achieved
Physical simulation	Achieved
Key frame animation	Achieved
Collision detection	Achieved
Multi-pass rendering	Achieved
Sound effects	Achieved
Gamer Intelligence	Achieved
Level of Detail (MipMapping)	Achieved

Table 1 Game Concept Objectives

Additionally, the following components were also added to the game:

- Particle Engine
- Level Culling – Similar to frustrum culling but specific to this game. Objects on other levels are not draw while still physically affecting the gamestate.

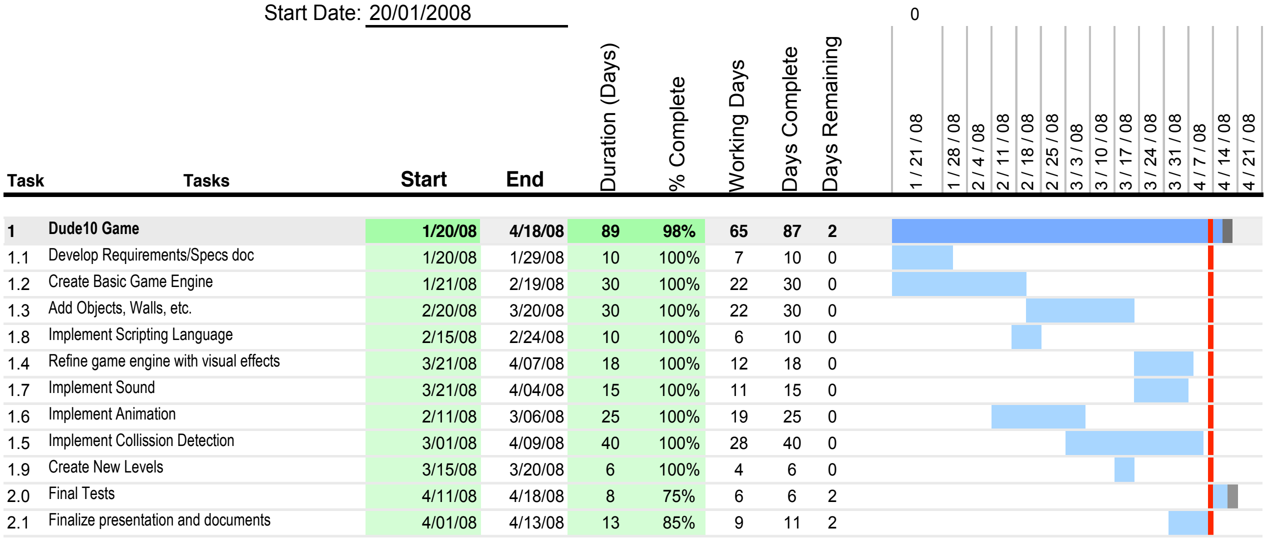
2.2 Milestones

Dude10 was designed to be a small-sized computer game that was to be developed over a 12 week period by a team of five amateur game developers. The sections described in the technical specification (MacWilliam, Dude10-Technical Specification 2008) were split amongst the developers with the intention of integrating them one by one to create the final release candidate. Table 2 is a summary of these milestones.

Date	Milestone
30/01/2008	Project Inception
19/02/2008	Completion of Basic Engine
30/03/2008	Additional Levels implemented
12/04/2008	Game completed
18/04/2008	Release Candidate Demonstrated

Table 2 Milestones Summary

Below is an updated Gantt chart reflecting the actual order of tasks worked on, with the start and end dates for each component. All tasks and their respective numbers have been kept the same as those found in Section 8.2 of the Game Concept (MacWilliam, Dude10 Game Concept 2008). The major differences to this Gantt chart and the one originally proposed is in the lengths of time needed to complete each task, and a much larger delay in the start date for the sound and visual effects tasks.



Start Date: 20/01/2008

Task	Tasks	Start	End	Duration (Days)	% Complete	Working Days	Days Complete	Days Remaining
1	**Dude10 Game**	**1/20/08**	**4/18/08**	**89**	**98%**	**65**	**87**	**2**
1.1	Develop Requirements/Specs doc	1/20/08	1/29/08	10	100%	7	10	0
1.2	Create Basic Game Engine	1/21/08	2/19/08	30	100%	22	30	0
1.3	Add Objects, Walls, etc.	2/20/08	3/20/08	30	100%	22	30	0
1.8	Implement Scripting Language	2/15/08	2/24/08	10	100%	6	10	0
1.4	Refine game engine with visual effects	3/21/08	4/07/08	18	100%	12	18	0
1.7	Implement Sound	3/21/08	4/04/08	15	100%	11	15	0
1.6	Implement Animation	2/11/08	3/06/08	25	100%	19	25	0
1.5	Implement Collission Detection	3/01/08	4/09/08	40	100%	28	40	0
1.9	Create New Levels	3/15/08	3/20/08	6	100%	4	6	0
2.0	Final Tests	4/11/08	4/18/08	8	75%	6	6	2
2.1	Finalize presentation and documents	4/01/08	4/13/08	13	85%	9	11	2

Figure 2.1 Final Gantt Chart

2.3 Project Results

This section will explore the current state of the game. A brief discussion regarding some key features of the game will be provided along with some screenshots.

2.3.1 Features

The selection menu screen that was discussed in section 2.2.2 of the *Functional Specification* and 6.1.2 of the *Technical Specification* is shown in Figure 2.2. Because a game is not in progress the "continue" option is not selectable.

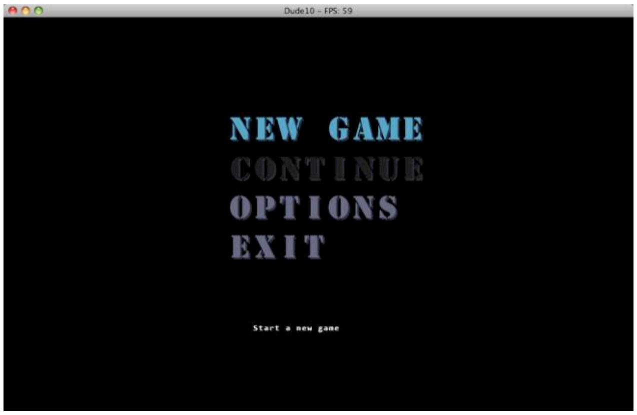
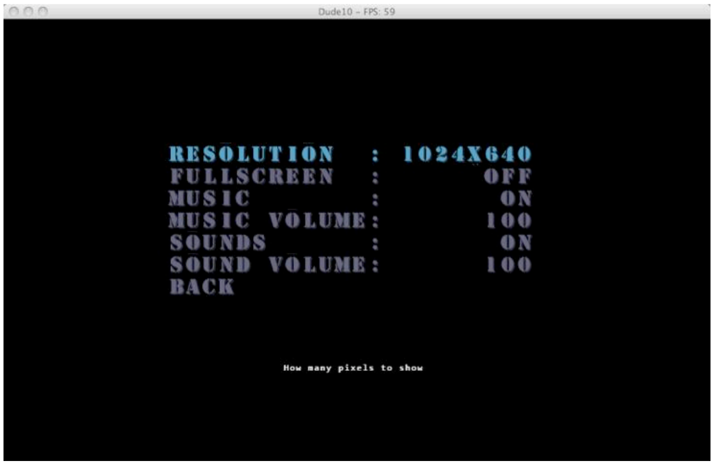


Figure 2.2 Main Menu Screenshot

Figure 2.3 shows the option menu screen discussed in section 2.2.3 of the *Functional Specification* and section 6.1.6 of the *Technical Specification*. All options can be changed with the preferences being saved automatically.

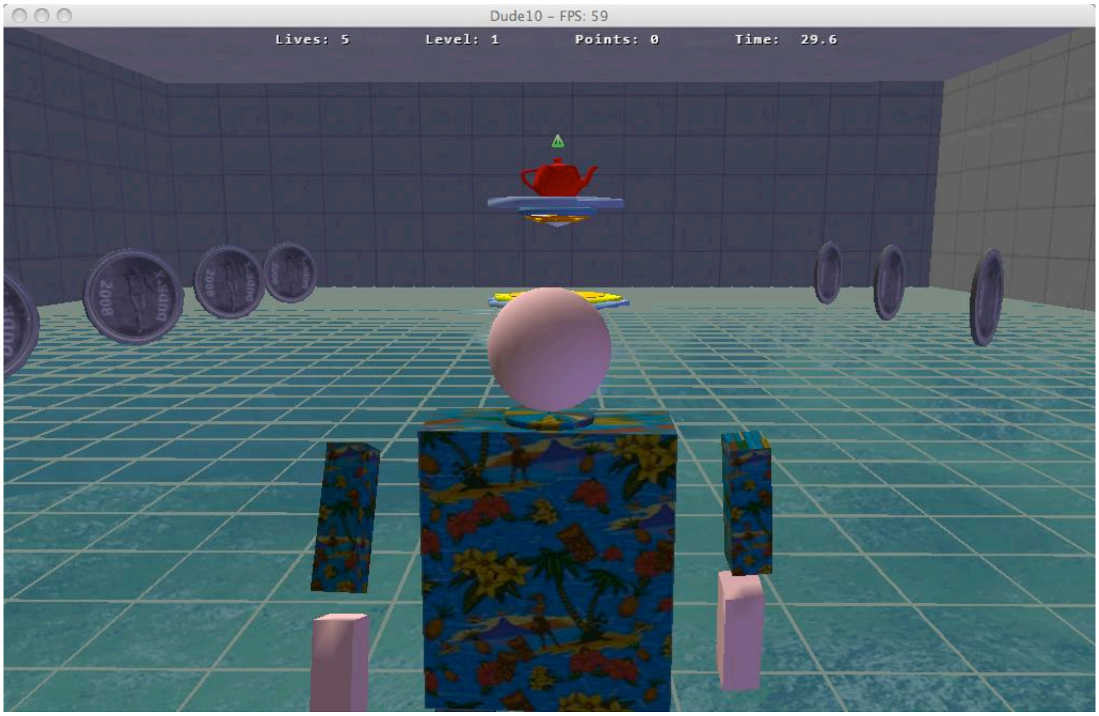


```
RESOLUTION    :  1024X640
FULLSCREEN    :       OFF
MUSIC         :        ON
MUSIC VOLUME  :       100
SOUNDS        :        ON
SOUND VOLUME  :       100
BACK

How many pixels to show
```

Figure 2.3 Options Menu Screenshot

Figure 2.4 is a screenshot of the first level of the game. This is what the gamer will see when they first click ``New Game`` on selection menu screen. The HUD is displayed along the top of the screen. The Dude is front and center in the Hawaiian shirt. Directly in front of the Dude on the other side of the room is an up elevator (signified by the red teacup). There are collectable coins to the left and right of the Dude. Notice the transparent floor with the terrain skybox visible.



Lives: 5 Level: 1 Points: 0 Time: 29.6

Figure 2.4 First Level Screenshot

Figure 2.5 shows 2 ghost Dudes and the currently played Dude. Notice how the lives have been decreased to 3, because 2 ghosts are active, which means the gamer has already played through 2 rounds of 45 seconds. Although it is hard to see the ghost Dude in the rear has just collected a coin and the particle effect is shown as the coin disappears.

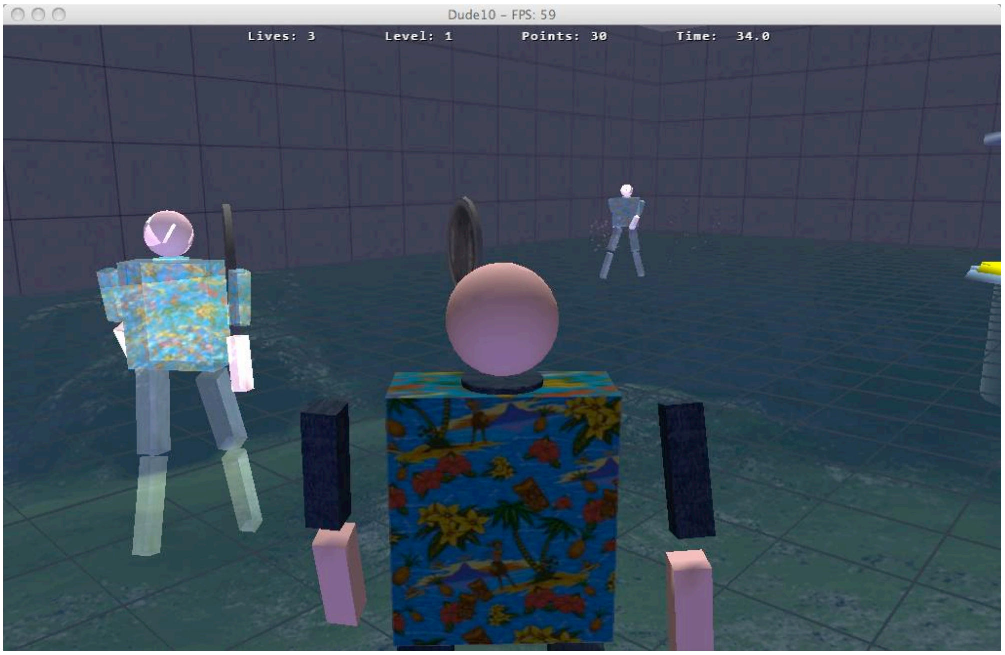


Figure 2.5 Ghost Dudes and Particle Effect Screenshot

Figure 2.6 and Figure 2.7 show the second level at two distinct periods in time (22.3 and 15.8 seconds). This is to show the clouds skybox is rotating while the terrain skybox remains in the same position. These figures also show a button the gamer has to walk over to open the up elevator to the 3rd level (the elevator with the red teacup).

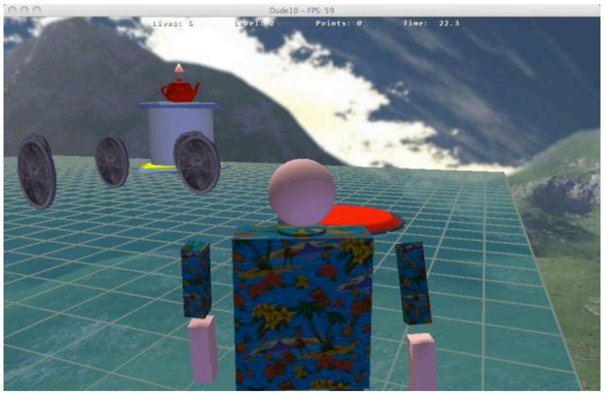


Figure 2.6 Cloud Screenshot #1

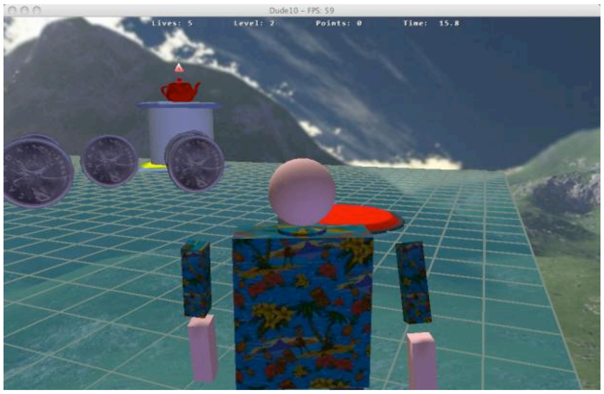


Figure 2.7 Cloud Screenshot #2

Figure 2.8 and Figure 2.9 show a clickable button found on level 4. Figure 2.9 shows the button when the gamer walks up to it for the first time. Figure 2.10 shows the same button that has been clicked 13 times. At this same point in the next round the now ghost Dude will have clicked the button to 87, but if the current Dude for that round was also clicking the same amount the result displayed would be 74.

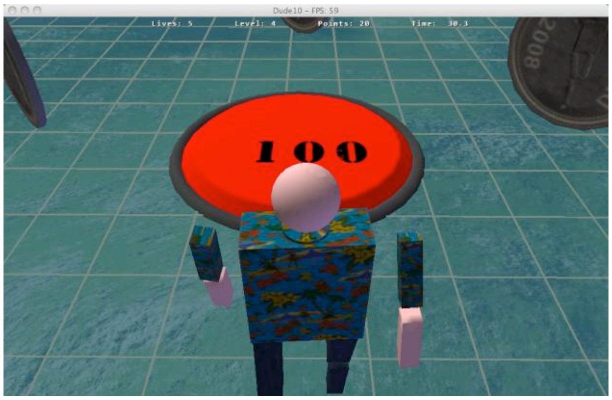


Figure 2.8 Clickable Button Screenshot #1

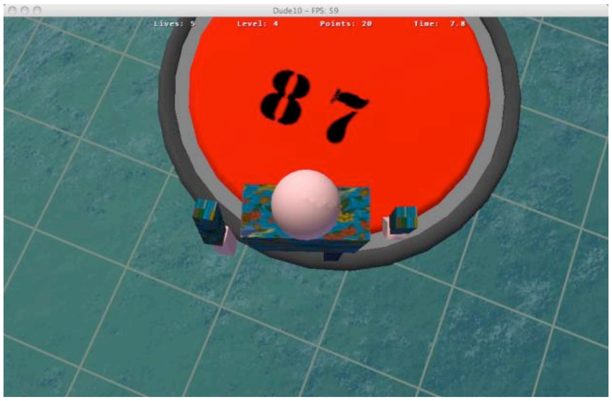


Figure 2.9 Clickable Button Screenshot #2

Figure 2.10 is a screenshot of the End of Game Screen discussed in section 6.1.5 of the *Technical Specification.* It is displayed when all the lives have run out, which occurs when the time reaches 0 on the Dude's last life.

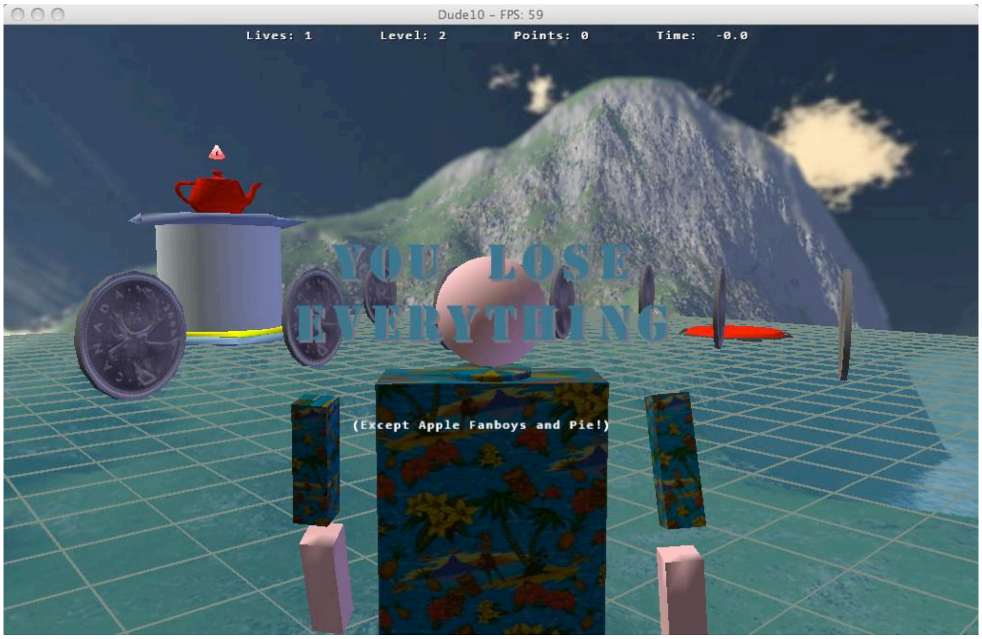


Figure 2.10 Lose Game Screenshot

2.3.2 Bugs

For a list of bugs that have been discovered, as well as the status for each bug, please see http://eece478.pbwiki.com (invite key: 'tyler')

3 Project Performance

In terms of development statistics Dude10 achieved the following:

SLOC: 22452

Number of Files: 209

Average File Size: 107.4 lines

The development of Dude10 increased steadily over time as can be seen in Figure 3.1.

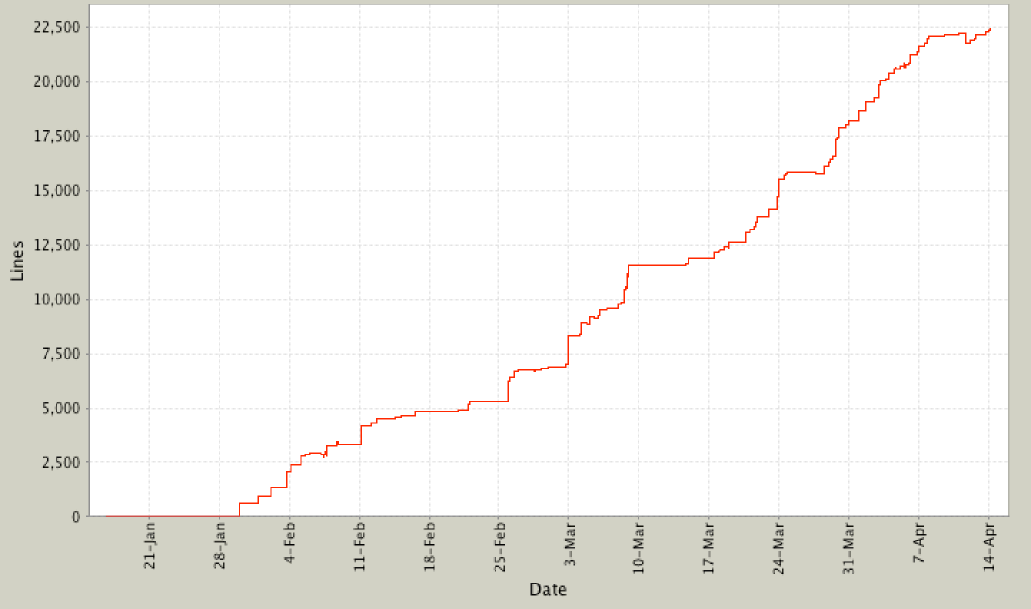


Figure 3.1 SLOC vs Time

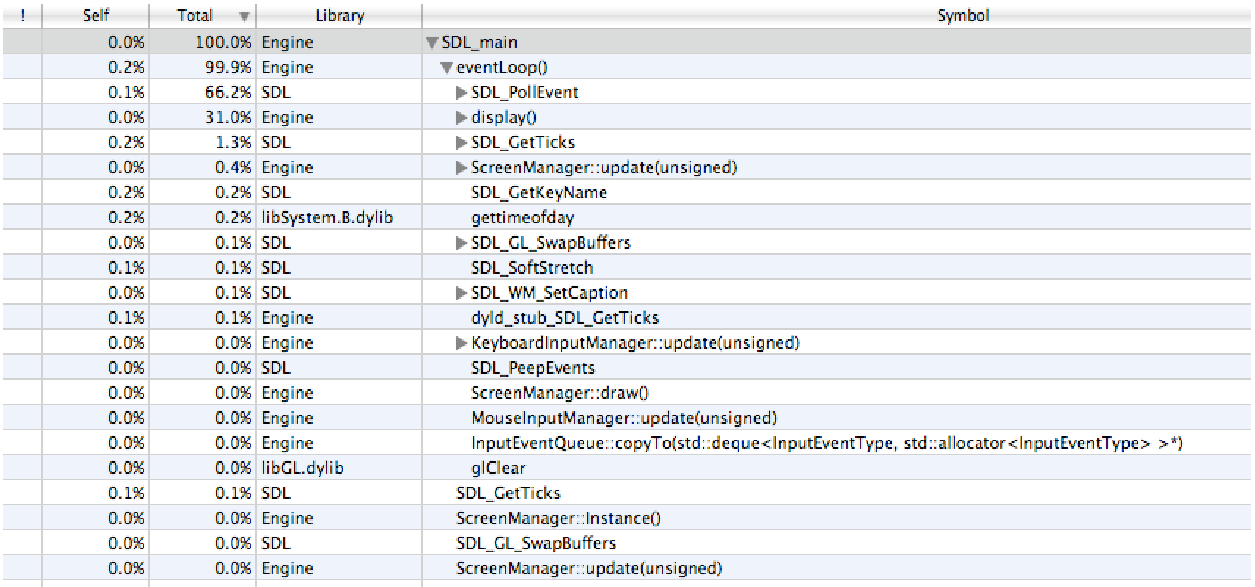
Below is a table of the various platforms the Dude*X was run on, along with the average frames per second, as well as the amount of system memory and CPU time used.

Platform	FPS	System Memory	CPU
IBM Thinkpad **Core Duo processor T2400 @1.86Ghz,** **1G memory,** **Windows Vista Business** **ATI Mobility Radeon X1400**	20	73MB	50%
Macbook Pro **Core 2 Duo 2.2GHz** **2GB Memory** **4 GB L2 Cache**	59	89MB	~97%
Macbook **2GHz Core duo** **1 GB 667 MHz DDR2 SDRAM**	20	90MB	~99%

Table 3 Game Performance

To determine which parts of the code are run most often a 30 second sample of normal game play was run through *Shark* (Apple Inc. 2004) on OSX 10.5 to generate a time profile. The Engine library which encompassed the entire code base of our game used 58.6% of the processing power of the game. The rest was devoted to the mach_kernel library.

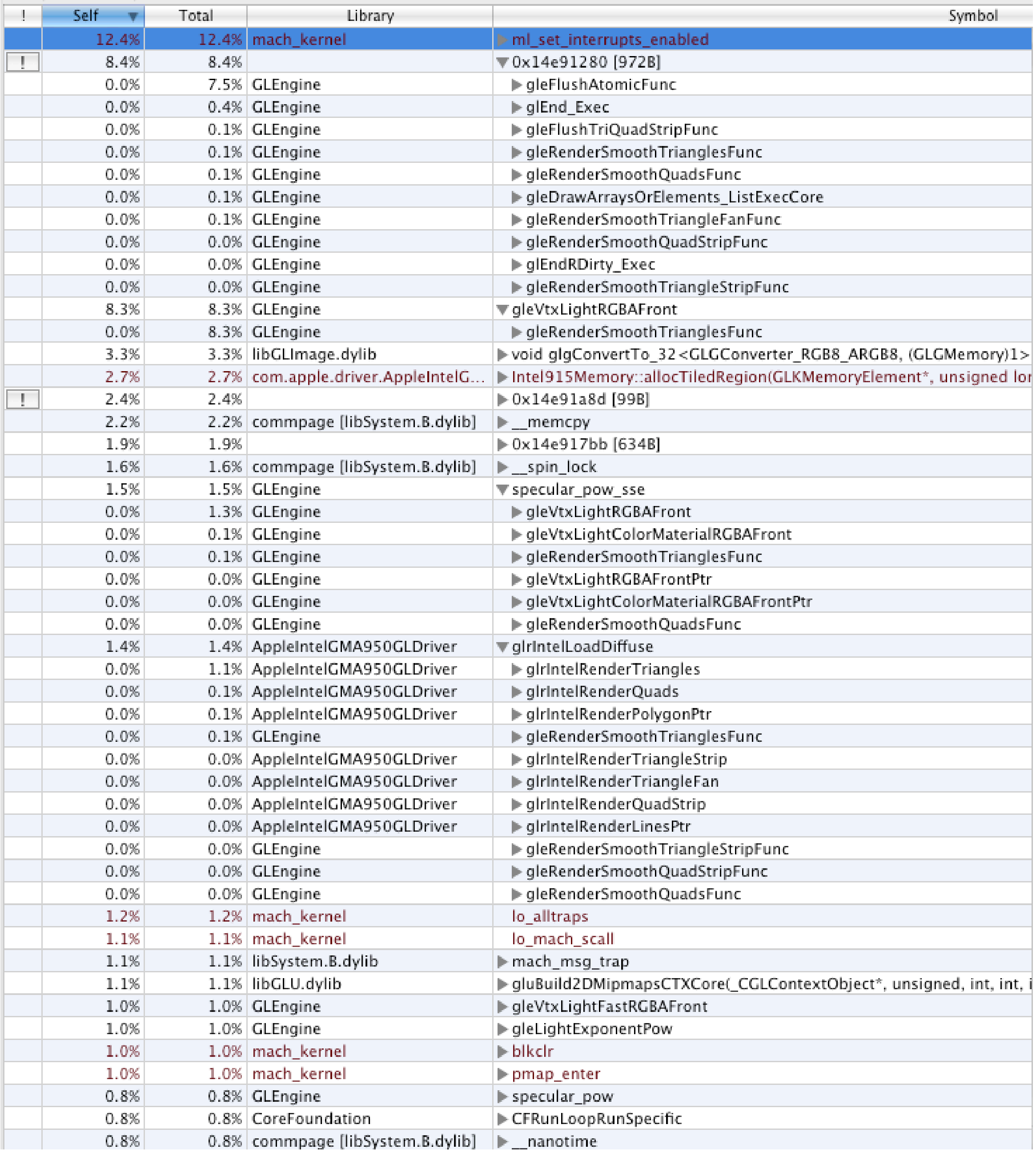
Figure 3.2 is the result of performing some data mining on the Engine library. By focusing on the SDL_main class, which contains the main function of the program, it is obvious that a majority of the program is spent inside the eventLoop function, as expected. Outside and inside of the eventloop function the a lot of time is spent getting the instance of the ScreenManager class and updating the ScreenManager. This too is expected, because ScreenManager controls which screens are displayed and is needed to redraw the screen. Nothing appears to be out of the ordinary, but further investigation will be required in order to locate and optimize the code.



!	Self	Total ▼	Library	Symbol
	0.0%	100.0%	Engine	▼SDL_main
	0.2%	99.9%	Engine	▼eventLoop()
	0.1%	66.2%	SDL	▶SDL_PollEvent
	0.0%	31.0%	Engine	▶display()
	0.2%	1.3%	SDL	▶SDL_GetTicks
	0.0%	0.4%	Engine	▶ScreenManager::update(unsigned)
	0.2%	0.2%	SDL	SDL_GetKeyName
	0.2%	0.2%	libSystem.B.dylib	gettimeofday
	0.0%	0.1%	SDL	▶SDL_GL_SwapBuffers
	0.1%	0.1%	SDL	SDL_SoftStretch
	0.0%	0.1%	SDL	▶SDL_WM_SetCaption
	0.1%	0.1%	Engine	dyld_stub_SDL_GetTicks
	0.0%	0.0%	Engine	▶KeyboardInputManager::update(unsigned)
	0.0%	0.0%	SDL	SDL_PeepEvents
	0.0%	0.0%	Engine	ScreenManager::draw()
	0.0%	0.0%	Engine	MouseInputManager::update(unsigned)
	0.0%	0.0%	Engine	InputEventQueue::copyTo(std::deque<InputEventType, std::allocator<InputEventType> >*)
	0.0%	0.0%	libGL.dylib	glClear
	0.1%	0.1%	SDL	SDL_GetTicks
	0.0%	0.0%	Engine	ScreenManager::Instance()
	0.0%	0.0%	SDL	SDL_GL_SwapBuffers
	0.0%	0.0%	Engine	ScreenManager::update(unsigned)

Figure 3.2 Engine Library Code Breakdown

Figure 3.3 is for reference only and is a breakdown of the various calls to the OpenGL library during a typical 30 second game play sample. As can be seen a lot of time is spent rendering quads and triangles, as is expected.



!	Self ▼	Total	Library	Symbol
	12.4%	12.4%	mach_kernel	▶ ml_set_interrupts_enabled
!	8.4%	8.4%		▼ 0x14e91280 [972B]
	0.0%	7.5%	GLEngine	▶ gleFlushAtomicFunc
	0.0%	0.4%	GLEngine	▶ glEnd_Exec
	0.0%	0.1%	GLEngine	▶ gleFlushTriQuadStripFunc
	0.0%	0.1%	GLEngine	▶ gleRenderSmoothTrianglesFunc
	0.0%	0.1%	GLEngine	▶ gleRenderSmoothQuadsFunc
	0.0%	0.1%	GLEngine	▶ gleDrawArraysOrElements_ListExecCore
	0.0%	0.1%	GLEngine	▶ gleRenderSmoothTriangleFanFunc
	0.0%	0.0%	GLEngine	▶ gleRenderSmoothQuadStripFunc
	0.0%	0.0%	GLEngine	▶ glEndRDirty_Exec
	0.0%	0.0%	GLEngine	▶ gleRenderSmoothTriangleStripFunc
	8.3%	8.3%	GLEngine	▼ gleVtxLightRGBAFront
	0.0%	8.3%	GLEngine	▶ gleRenderSmoothTrianglesFunc
	3.3%	3.3%	libGLImage.dylib	▶ void glgConvertTo_32<GLGConverter_RGB8_ARGB8, (GLGMemory)1>
	2.7%	2.7%	com.apple.driver.AppleIntelG…	▶ Intel915Memory::allocTiledRegion(GLKMemoryElement*, unsigned lor
!	2.4%	2.4%		▶ 0x14e91a8d [99B]
	2.2%	2.2%	commpage [libSystem.B.dylib]	▶ __memcpy
	1.9%	1.9%		▶ 0x14e917bb [634B]
	1.6%	1.6%	commpage [libSystem.B.dylib]	▶ __spin_lock
	1.5%	1.5%	GLEngine	▼ specular_pow_sse
	0.0%	1.3%	GLEngine	▶ gleVtxLightRGBAFront
	0.0%	0.1%	GLEngine	▶ gleVtxLightColorMaterialRGBAFront
	0.0%	0.1%	GLEngine	▶ gleRenderSmoothTrianglesFunc
	0.0%	0.0%	GLEngine	▶ gleVtxLightRGBAFrontPtr
	0.0%	0.0%	GLEngine	▶ gleVtxLightColorMaterialRGBAFrontPtr
	0.0%	0.0%	GLEngine	▶ gleRenderSmoothQuadsFunc
	1.4%	1.4%	AppleIntelGMA950GLDriver	▼ glrIntelLoadDiffuse
	0.0%	1.1%	AppleIntelGMA950GLDriver	▶ glrIntelRenderTriangles
	0.0%	0.1%	AppleIntelGMA950GLDriver	▶ glrIntelRenderQuads
	0.0%	0.1%	AppleIntelGMA950GLDriver	▶ glrIntelRenderPolygonPtr
	0.0%	0.1%	GLEngine	▶ gleRenderSmoothTrianglesFunc
	0.0%	0.0%	AppleIntelGMA950GLDriver	▶ glrIntelRenderTriangleStrip
	0.0%	0.0%	AppleIntelGMA950GLDriver	▶ glrIntelRenderTriangleFan
	0.0%	0.0%	AppleIntelGMA950GLDriver	▶ glrIntelRenderQuadStrip
	0.0%	0.0%	AppleIntelGMA950GLDriver	▶ glrIntelRenderLinesPtr
	0.0%	0.0%	GLEngine	▶ gleRenderSmoothTriangleStripFunc
	0.0%	0.0%	GLEngine	▶ gleRenderSmoothQuadStripFunc
	0.0%	0.0%	GLEngine	▶ gleRenderSmoothQuadsFunc
	1.2%	1.2%	mach_kernel	lo_alltraps
	1.1%	1.1%	mach_kernel	lo_mach_scall
	1.1%	1.1%	libSystem.B.dylib	▶ mach_msg_trap
	1.1%	1.1%	libGLU.dylib	▶ gluBuild2DMipmapsCTXCore(_CGLContextObject*, unsigned, int, int, i
	1.0%	1.0%	GLEngine	▶ gleVtxLightFastRGBAFront
	1.0%	1.0%	GLEngine	▶ gleLightExponentPow
	1.0%	1.0%	mach_kernel	▶ blkclr
	1.0%	1.0%	mach_kernel	▶ pmap_enter
	0.8%	0.8%	GLEngine	▶ specular_pow
	0.8%	0.8%	CoreFoundation	▶ CFRunLoopRunSpecific
	0.8%	0.8%	commpage [libSystem.B.dylib]	▶ __nanotime

Figure 3.3 GL Library Code Breakdown

4 Recommendations

Overall the developers are quite pleased with the results of Dude*X, but there is still some room for improvement. To make the game more realistic shadows should be implemented for all characters and objects (except for ghost dudes, because they are ghosts). The ghost Dudes are currently not animated, and even though they float around like ghosts the developers feel that in the future they should be animated.

In the original planning session of the game more complicated levels had been brainstormed, but they could not be implemented due to the time frame. These levels included moving boxes to cross a chasm, pulling on ropes to open doors, as well as the use of switches along with the buttons that have already been implemented. These additions could be implemented in the existing 7 level structure of the game, or they could be added as new levels. Of course, if they are new levels the life for each Dude will need to be increased.

Finally, as mentioned in section 3, now that the code base has been completed and is relatively mature the focus should be on optimizing the code to obtain a better frame rate or decrease the amount of memory or processor power needed.

Appendix A: Development Breakdown

The breakdown of development was as follows:

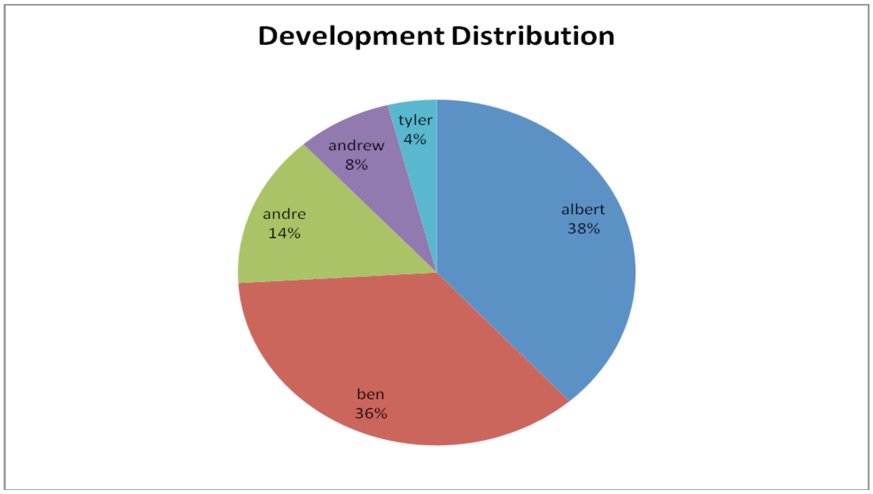


Figure A.1 Development Distribution

For detailed work logs and other statistics please see the StatSVN page for the Dude*X project that can be found at https://exit.ath.cx/statsvn. For more information about the resources, project ideas and practices we used please see http://eece478.pbwiki.com (pass key: ‘tyler’)

To summarize the information found on both sites, the key features developed by the programmers were:

Tyler MacWilliam

- Documentation
  - Wrote up all documentation except for sections 4,5 and 7.1 of the Technical specification and Appendix A of this document (everyone wrote their individual parts)
  - It was decided that one person should be in charge of the documentation in order to maintain consistency and give it the priority it needed (25% of the final grade)
- Sound
  - Create a sound manager class that loaded and played sound effects and background music. Both types of sounds were treated differently and could be called either using a unique id number or the filename

  - Modified and added code from LuaObjMgr class to create the LuaSoundMgr class which brought sound functionality created in the sound Manager class to Lua. These commands (i.e. playAudioID(letsGo)) could be called through the Lua Console or in Lua scripts
  - Implemented the correct function calls at appropriate places in the code to call the sounds (i.e. change background music for each new level)
  - Recorded custom sounds (coin pick up, Dude voiceover, elevators) and found royalty free loops and effects that could be used when the sounds couldn't be created

- Coin
  - Wrote the base code needed to place and draw coins through Lua. Ben helped with drawing the textures correctly and rotating them. Albert helped figure out how the coins could be reset each time a new round starts so either ghost Dudes or the current Dude could collect them

Ben Randall

- Main graphics engine functionality
  - Modified Vertex classes (created by Neema Teymory) and created simple Quaternion class to be used for rotations.
  - Implemented cameras and camera manager classes. All camera functionality is manually implemented (i.e. no GLU methods used). Quaternions used to manage rotations.
- Game State Management
  - ScreenManager classes to handle the screen stack. Supports multiple screens overlayed overtop of each other (HUDScreen overtop of GameScreen).
  - Screen Stack also used to handle transitions between screens using FadeScreen.
  - GameEvent system supports 'asynchronous' event sources and bubbling events to communicate between various screens.
- Game Object and Animation System
  - IControllable/IController base class to provide a way to easily manipulate game items. This is used to move/rotate ingame items. Any IControllable can be controlled by any IController. This is used by Lights, Cameras, Models, etc.

  - Keyframeable animation system allows basic tweening of IControllable properties with ease.

- Additional 3DS Model loading work
  - Mostly loading 3DS animations and converting to internal animation system.
  - Extended animation system to allow animating individual model meshes
- Dude Implementation
  - Created Dude Class to handle the 3DS Dude Model I created along with the 3rd person camera and controller.
  - Created Dude Controller. This is a combination of some existing KeyboardController stuff along with MouseController for easy management.
- Lighting System
  - LightingManager and Light classes to more easily interface with OpenGL lighting
- Font System
  - Loading of font textures (also generated by our group) and manually create display lists for drawing the fonts. Handling of all setup and teardown involved in that as well. No existing font system functionality used.
- In-Game Console
  - Access Lua functionality and view Lua output. Allows easy modification of game properties at runtime via LuaGL (Lua bindings for OpenGL).
  - Command scroll back for easy modification and re-execution of previous commands

Albert Sodyl

- Physics Engine
  - All collision detection and response, bounding volumes and physical objects and anything else in the Physics Engine.
  - Later add support for generating events based on collisions.
- Resource Manager

- Cross-platform abstraction layer to load and read files, supporting multiple search directories for resources and the ability to switch game types for the game engine.

- Background Sky
    - Created the initial sky model (non-rotating).
- Portability
    - Modified 3dsftk library to support other platforms better, including fixes for Big Endian systems and 64-bit machines.
    - Created PlatformUtils, UnixUtils, MacUtils, and WinUtils to deal with platform differences for certain functionality.
    - Keep code constantly workable on Linux and Mac, including library differences, and keep the Linux makefile and Xcode project up-to-date.
- Menus
    - Expanded original title screen to several screens, including a main menu and options screen with controllable items that can be selected, unselected, and disabled.
- Preferences
    - Implement ability to retrieve and set preferences through the game interface, with an abstraction layer of how the preferences are saved. These can be used for many options, including top scores.
- Event Handling and Recording
    - Created an event-driven system with an event manager and plug-in listeners for “push” based events that can be recorded.
    - Recording both “poll” and “push” events and replaying them for each round.
- Particle Effect System
    - Basic particle effects that vary depending on the configuration.
    - Add particle effect manager that controls all current particle effects.
- Mouse Support
    - Added ability to control the game via the mouse in addition to the keyboard.

Andre Soesilo

- Implemented Lua functionality:
  - Enable addition, removal and modification of all available models (e.g. elevator, button, walls) in real-time. Modifications of models include changing the position, the rotation and the size of the models.
  - Loading and saving the models
  - Enable addition and removal of bounding volumes for physics engine
  - Added functionality to execute Lua methods from Console
- Created and added elevator models
  - Created the elevator model in 3DS Max Studio
  - Added the elevator model to the game and its properties (e.g. activated/de-activated, to next/previous level)
- Created and added button models
  - Added the button model (created by Ben Randall) to the game and implemented its behaviour (e.g. pressed/released, which elevator it is associated with, number of mouse clicks needed to be activate the elevator, toggle/switch button)
- Implemented multiple life and multiple levels functionality to the game
  - Transition between levels
  - Transition to the next life
  - Created the levels (i.e. location of each model, what each level does)
  - Created DudeManager to manage the current dude and keeping track of the older dudes
  - Created DrawManager to give flexibility in adding models to draw in the level and knowing which models to draw for each level
- Added the Skybox (not BackgroundModel)

  - Modified BackgroundModel to be rotatable

- Kept track of the team to make sure that we were in the right track and that everyone was progressing fine with their tasks

Andrew Thompson

- Implemented Toolkit 3DS File Loader
  - Parsed 3DS file to obtain model vertices, normals, colours and materials
  - Also obtained animation key frames for certain models (i.e. The Dude and Elevator)
  - Interpolated key frame data for translation and scaling.
  - Created display lists for model components
- Game Screens
  - Designed HUD screen (Heads Up Display) so that timing, lives, level…etc information is displayed to the gamer
  - Implemented the end of game screen to trigger when time ran out
- Helped with the continual porting of our game to the MAC OS X platform
  - Modifying project settings and keeping the project up to date