**Introduction**

Welcome to Unknown Unicycler! This is a 3D unicycling game where the player must collect all of the coins to progress to the next level.

This game is made in C++ using OpenGL for rendering graphics. FreeGLUT (an OpenGL Utility Toolkit) to handle the gritty side of “creating windows, initializing OpenGL contexts, and handling input events” (freeglut.sourceforge.net), and was built in Visual Studio 2019.

The object-oriented game is composed of five key components: Player, World, Camera, Keyboard, and Game. Each object is created once but allows for scalability in hypothetical later releases of Unknown Unicycler. For instance, multiple Player objects can be created for multiplayer functionality.

I was able to complete this project thanks to my Spring 2017 semester abroad at the University of Canterbury in Christchurch, New Zealand. There I took Computer Graphics COSC363 which focused on graphics theory (such as transformations, lighting, projections, etc), graphics programming (including the OpenGL API, application development, animations), and geometric algorithms (ray tracing and surface design). I have previously created 3D animations with basic keyboard input and always imagined making a full game at scale. Unfortunately, this proved harder than anticipated as OpenGL is generally advised to be used purely for educational purposes. All of my internet research strongly suggested using a professional game engine.

**Installation Instructions**

All necessary files are be contained in the .zip provided. It is a copy of my “Debug” folder from the Visual Studio 2019 project. Extract it to a convenient location, run Unknown.exe and enjoy.

|  |  |
| --- | --- |
| Unknown4.exe | The game executable |
| freeglut.dll | OpenGL Utility Toolkit library |
| glew.dll | OpenGL Extension Wrangler library |
| Unknown4.ilk | Incremental linker file (by Visual Studio) |
| Unknown4.pdb | Debugging database (by Visual Studio) |
| glewinfo.exe | Creates ‘glewinfo.txt’ (by Visual Studio) |
| visualinfo.exe | Creates ‘visualinfo.txt’ (by Visual Studio) |

To run the game, execute “Unknown4.exe.”

**User Controls**

|  |  |
| --- | --- |
| Player controls | |
| W | Move forwards |
| S | Move backwards |
| A | Turn left |
| D | Turn right |
| [spacebar] | Jump |
| Y | Progress to next level |

|  |  |
| --- | --- |
| Camera controls | |
| I | Move forwards |
| K | Move backwards |
| J | Turn left |
| L | Turn right |
| U | Move down |
| O | Move up |

|  |  |
| --- | --- |
| Admin controls | |
| T | Toggle information display |
| R | Reset the level |
| [arrow up] | Increase FPS |
| [arrow down] | Decrease FPS |
| [number keys] | Set level (0-2) |

**Design**

There are five main objects to orchestrate this project. The Player, World, and Camera class all contain individual information pertaining to their respective objects, such as positions and rendering functions. The Keyboard class records which keyboard keys are pressed down or released. The Game class contains and manages the aforementioned objects while organizing the game’s flow of logic.

There are two other auxiliary classes: Commons.h and Play.cpp. Commons.h contains the structs used in the World class. Play.cpp contains the main() function, initializes OpenGL and establishes the event listeners.

The variables are named using camelCase. Underscores are only used to denote x, y, or z, of variables such as position or velocity.

*Challenges*

One design challenge was determining when or where to handle keyboard input. Ideally, all keyboard presses would be located in the same file so keys can easily be assigned. However, I was not able to accomplish this. Instead, the controls are scattered about the project files, with an attempt to keep them relatively close within each area.

Another challenge is object deconstructors. Because each object exists once and for the lifetime of the program, the destructors exist but are empty.

The biggest challenge which has yet to be resolvedis the issue of box/star collision. For some unknown reason there is a loss of data in the main loop (Game::display). While I define the boxes/stars in the World class, they should be accessible to the Game class (World is a member of Game). Yet, when it comes to checking collision, the array of boxes/stars in World contains undefined values. I have tried explicitly passing the arrays by reference, with and without , making the arrays global, and unfortunately was never able to find a solution.

*Play.cpp*

Designing this class was one of the biggest challenges as it establishes OpenGL and event listeners without containing any substance for the game itself. Using OpenGL requires registering callback functions. My previous work with OpenGL was limited to one .cpp file, so my first idea was to assign the object’s functions (e.g. Player::draw) here. Unfortunately, this did not work because OpenGL is a C API and therefore object functions cannot be registered as callback functions. My solution is to create ‘wrapper’ classes which would be registered for OpenGL’s callback functions and call my object’s functions from there.

Callback functions include:

* Display
* Keyboard down
* Keyboard up
* Special keyboard presses
* Window reshape

The Play class contains one Game object, allowing the Play class and its wrapper functions to call Game’s functions. This methodology allows OpenGL to have its callback functions registered successfully. Within these wrapper classes I call the functions of Game that I originally tried to register as callback functions. The necessary parameters in the wrapper functions can easily be passed to Game’s functions. For instance, the keyboard press event listener in Play will call Game’s keyboard press function including the key pressed as a parameter.

This class is the entry point for OpenGL and was my biggest hurdle for this project. Until I understood how to organize this code, I was unable to make any other progress as it is the true foundation for this project. My meetings with Professor Frees solidified my train of thought as he helped me understand how to manage the C-style API functionality of OpenGL with my object-oriented C++ code.

*initialize*

This function will establish the background color, lighting, and the camera’s frustrum.

*displayWrapper*

This function tells the game to draw a new scene to the window; it is essentially the animation loop for the game. In addition to passing the function from this wrapper class to Game’s display function, this display wrapper will also regulate the upper framerate of the game. Play.cpp uses clock\_t and time\_t objects to time how long the scene rendering takes and wait the appropriate amount of time before drawing the next frame.

*keyboardDownWrapepr*

This function will call the Keyboard’s *keyboardDown* function, passing the key pressed as a parameter.

*keyboardUpWrapper*

This function will call the Keyboard’s *keyboardUp* function, passing the key pressed as a parameter.

*specialWrapper*

This function will call the Keyboard’s *specialKeys* function, passing the key pressed as a parameter. Special keys include the Function keys (F1-F12), Arrow Keys, Page Up, Page Down, Home, End, and Insert. This function will act upon two key inputs, Arrow Up and Arrow Down, to modify the game’ framerate. I included this here because the Keyboard class does not have access to Game’s variables.

*reshape*

This function will maintain proper aspect ratio when the Game window is resized. This allows the user to play the game at any window size they like, instead of the default 1000px x 655px.

*main*

This function will initialize OpenGL, establish callback functions, and begin the main processing loop. No code after here is executed.

*Game.cpp*

The Game class is the body of my project. It contains all other objects (Player, Camera, World, Keyboard) and orchestrates the game’s flow and logic. By containing the other classes, it manages the others’ information. The Game class acts as the conductor of this digital orchestra.

*Game::Game*

Upon construction the game will initialize GLUT and the window for the game. It will also set the level to load first.

*Game::initWindow*

This function establishes the buffer system to load matrices before rendering them and creates the window that the Game will render in.

*Game::display*

The Game’s display function acts as the engine of this project. It handles the animation loop, collision detection, overlays text on the screen, some keyboard input, and renders the scene.

Its first act of business is to establish the light in the world, which is omni-directional (it shines in all directions, from one position, like the sun).

The second act of business is to direct the Camera and Player to move based on the keys currently being pressed. Because the three aforementioned objects are members of the Game class, it passes the keys pressed (from the Keyboard class) to the Camera and Player. This will update their respective positions.

Next, the Game object will check for any collisions with stars, boxes, and the floor. These functions will update the Player’s position accordingly. Similarly, Game will check to make sure that the Player stays within the bounds of the world.

The display function will then check if it should reset or change the level by checking if the appropriate keys are pressed.

After the possibility of updating the positions of the Player or Camera, the Game will get ready to load all aspects of the game to the screen. This includes the Player, World, and text. Loading these aspects will load them to a buffer, which is rendered to the screen at the conclusion of Game::display.

*Game::checkStarCollisions*

The method of 3D collision detection I used is known as axis-aligned bounding boxes (<https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection>). This function is the first in collision detection and checks if a point is inside a box. It works by running through each of the stars in the World and assigning a temporary box where the star’s point is. Once assigned, Game will check if the Player’s point in 3D space is inside (edges inclusive) this temporary box. If so, the star will be hidden.

*Game::checkBoxCollisions*

This function is similar to *checkStarCollisions* by using axis-aligned bounding boxes for the boxes in the world and three points on the unicycle. For each box, the function checks if either the front, center, or rear part of the wheel is inside the respective box. If so, there are two possible routes: either the top of the box is low enough that the Player will ‘roll’ right onto it, or the Player is stopped before moving into the box. This function will also, if debugging mode is on, draw yellow pillars at the corner of each box and the three points on the Player which is helpful for debugging visually. Additionally, if debugging is on, it will draw text on the screen indicating which part of the wheel is in which box. Currently only a seemingly random selection of boxes will have collision detection implemented.

*Game::checkFloorCollisions*

This function is fairly straightforward by preventing the Player’s Y coordinate from dropping below 0.

*Game::isPointInBox*

This function uses the pseudocode from the aforementioned link to check if a given point (x, y, z) is inside the given box (edges inclusive).

*Game::drawString*

This function will draw a string onto the screen at a given position. It was one of the more challenging functions to develop as I have never worked with rending text outside of the console. To do so involves saving the state of the current matrix in buffer (the scene to draw) by wrapping the substantial content of this function in stack push/pop calls. Due to the limited nature of OpenGL, the text will always be a white 9px x 15px font (which scales with the window).

*Game::displayScore*

This function will get the score from the World class and draw it on the left-side of the screen using the *Game::drawString* function.

*Game::debugStats*

This function will draw the Player’s position, velocity, grounded status, and Game framerate on the screen using the *Game::drawstring* function.

*Game::resetLevel*

This function will call upon the Player’s, World’s, and Camera’s reset function.

*Game::toggleDebug*

This function will invert the debugging value.

The most apparent challenge of this class is the flow of game design. Determining how and when to move the Player and Camera used to live in this function but became extremely convoluted and I resolved this by moving them to their own respective classes. I had to be sure to update positions before drawing to the screen.

Another example of the challenges of the flow of logic is that the Player will not roll off of objects naturally. Instead, the fall needs to be triggered with a jump..

Here is one place where there is some keyboard input, but not all. Again, I would have preferred to keep this all in the same place but was unsure how. Because the Keyboard class is a member of the Game class, I could not call Game/Player/World functions from the Keyboard class.

Having the Game class own the other classes allows me to pass, for instance, World variables to the Player class when need be.

One problem I faced was the text not drawing on the screen. First, the scene was no longer rendering. I learned this was because of the different matrix modes that OpenGL uses, hence my need to store the matrices with a series of push matrix and pop matrix. Next, I was rendering the World and Player to the screen after the text, resulting in text being covered and unavailable.

*Player.cpp*

The Player class represents the user. It is a unicycle with no rider, which can face any direction and move forward, backward, and turn by either pivoting in place or turning while moving.

One major problem is the logic of Player movement. I organize it by first turning, then calculating the forward direction, then jumping and falling.

*Player::drawPlayer*

This function will call *Player::drawUnicycle* (which renders the Player) and both translate and rotate it to where it should be in the World.

*Player::move*

This function will first store the current position and direction of the Player which is used for collision prevention. It then checks for turn input which will be used for calculating the forward direction of the unicycle. It will then check for forward or backward input and move the position accordingly. This took some basic trigonometry to determine how far along the X and Z axis the Player should move depending on what direction they are facing. Next, the Player checks to see if it is grounded (on the floor or a box) and ideall*y* fall accordingly, this does not work**.** This function will also determine if the Player is moving at all and rotate the spokes at a velocity-dependent speed. Finally, it updates the position of the Player and its collision points (front wheel and back wheel) .

*Player::reset*

This function will set the Player’s position, velocity, and forward direction to 0.

*Player::keepInside*

This function will ensure that the Player’s position does not exceed the given World size. It also prevents the Player from falling through the floor.

*Player::revertPos*

This function will set the Player’s position to its last stored position and orientation. This is used for box intersection prevention.

*Player::drawUnicycle*

This function will utilize OpenGL to render the Player to the screen. It draws the tire, spokes, seat post, and seat in appropriate size and color.

*Player::drawBoundingBox*

This function will draw a wireframe box around the perimeter of the Player. It is useful for visual debugging.

*Player::getPos\_x*

This function will return the Player’s X coordinate.

*Player::getPos\_y*

This function will return the Player’s Y coordinate.

*Player::getPos\_z*

This function will return the Player’s Z coordinate.

*Player::setPos\_y*

This function will set the Player’s Y position.

*Player::setVel\_y*

This function will set the Player’s Y velocity.

*Player::getForward*

This function will return the Player’s forward direction in radians.

*Player::getGrounded*

This function will return the Player’s grounded status.

*Player::setGrounded*

This function will set the Player’s grounded status.

*Player::pos\_toString*

This function will return the Player’s 3-coordinate position as a string.

*Player::vel\_toString*

This function will return the Player’s 3-coordinate velocity as a string.

*Player::frontWheel\_toString*

This function will return the Player’s front wheel 2-coordinate position as a string.

*Player::back Wheel\_toString*

This function will return the Player’s back wheel 2-coordinate position as a string.

*World.cpp*

The World object represents the world, of course! While my original idea was to have a World object defined for each level, I managed to represent any level with one object due to the nature of OpenGL; because it has to draw the World with each frame, I draw whatever world should be loaded at that time.

Problems included keeping a list of boxes and stars, and the creative aspect of designing worlds. This was a reminder that I prefer backend development over front end.

*World::World*

This constructor will reset the level.

*World::~World*

This destructor does nothing.

*World::drawWorld*

This function will call the appropriate function to load the given level.

*World::drawZero*

This function will draw a flat world.

*World::drawOne*

This function will draw a floor with a box to roll onto, a box to jump onto, and 3 stars to collect.

*World::drawTwo*

This function will draw a floor and a series of boxes to jump to. They get taller as they go on, with a star to collect on top of the last one.

*World::drawThree*

This function will draw a floor and a rail for the user to ride with a star at the end.

*World::drawFloor*

This function will draw an orange-lined floor

*World::drawCheckeredFloor*

This function will draw a tile floor with an alternating color grid.

*World::drawBox*

This function will draw a box of a given size at a given location, while assigning it to the array of boxes in the world.

*World::drawPillar*

This function will draw a vertical pillar in a given location. It is useful for visual debugging.

*World::drawWall*

This function will draw a wall from a given position to a second given position. It does not have a bounding box, unlike the boxes.

*World::drawStar*

This function will draw a star at a given location, while assigning it to the array of stars in the world. There were plans to make this more star-shaped then coin-shaped, however I’ve grown to like the coin look. Changing every instance of ‘star’ to ‘coin’ seems to invite problems to my entire project.It will also rotate the stars here.

*World::getScore*

This function will count how many stars have been collected and return this value as an integer.

*World::score\_toString*

This function will return the score as a string.

*World::outlineBoxes*

This function will draw a vertical pillar at each corner of each box. It is useful for visual debugging.

*World::reset*

This function will set all stars as uncollected.

*Keyboard.cpp*

The Keyboard object has an array for all of the keys on the board. Each index in the array is its’ ASCII equivalent. In other words, I use characters as indices. It is pretty neat! Being a Boolean array, true represents a pressed key and false represents a released key.

*Keyboard::Keyboard*

Constructor – declare all keys as released.

*Keyboard::~Keyboard*

Destructor – do nothing

*Keyboard::keyboardDown*

This function will set the appropriate key in the array as down (pressed). It will terminate the program if the key pressed is Escape.

*Keyboard::keyboardUp*

This function will set the appropriate key in the array as up (released).

*Keyboard::specialKeys*

This function currently does nothing but can handle special keyboard presses.

*Camera*

This object is the viewpoint for the user. It has functionality to move and look around like the Player. This is a very straightforward object utilizing basic trigonometry to calculate forward direction and GLUT to establish the viewing matrix.

*Camera::move*

This function will see what buttons are pressed and adjust the position and orientation appropriately. The camera can look left, look right, move forward, move backward, move up, and move down.

*Camera::reset*

This function will set the position and orientation to its default values.

**Test plan and results**

First test: Play the game! Get all of the stars, watch the score counter increase, and progress through all of the levels. Roll onto short boxes and get blocked trying to roll into taller boxes. Notice how it is possible to wiggle into them and wiggle out. Get in and try to jump out – watch you sometimes pop to the top. Try to jump into the sides of boxes.

Although there is no way to monitor realtime performance, Unknown Unicycler includes an adjustable framerate to play at a speed to match you and your machine’s abilities. Use the arrow keys (up and down) to adjust the game’s speed.

Hit ‘T’ to toggle the debugging stats for a more visual understanding of the game’s mechanics. In the case of the phantom boxes, you’ll see that the Player’s position can in fact be inside the boxes but still not ‘collide’ with them.

Results

There is a massive bug that essentially makes the game unplayable. At some point in the code, between *Game::display, Game::check[Box/Star]Collisions,* and *Game::isPointInBox* there is lost data and the position of the [box/star] is nonsensical. This caused me to stop level development because there is no point in making more levels when the first few are unplayable.

Aside from this bug, the game runs as planned. The program supports simultaneous keyboard input, there is a sense of physics, there are multiple levels, and it all runs smoothly.

There is a short list of things I would have liked to implement but was unable to due to time restrictions: GUI, weather (including falling particles, a ‘wind’ force pushing the Player laterally, and different lighting), level completion countdown timer, high scores (creating a second window, reading/writing from an external file), and more diverse levels. Again, I was unable to implement these features due to time restrictions with my summer internship.

**Summary and Conclusion**

Making this game was both fun and frustrating, as any good computer science project. Making a 3D game from scratch is what I have been dreaming about since I started programming. Now that I have, I understand that there are many more efficient ways to accomplish this (i.e. using a professional engine).

My previous experience with OpenGL and GLUT established my confidence to accomplish this feat, however I greatly underestimated the importance of planning and design before coding. While I was able to render graphics without a problem, I struggled with connected the many pieces of this project into one coherent project.

Researching help for this project was a challenge because most game tutorials were more complicated than this relatively simple game. Meeting with Professor Frees (twice) helped me get my project moving during my initial setup period (i.e. before I created the Play class with callback wrapper functions).

Once I had my OpenGL established I began creating objects without a concrete plan of how to organize them. This resulted in creating only one of each object, which I would not have originally anticipated. For instance, I imagined I would create multiple World objects (one for each world) but found I could just as easily redefine one World object to act as any level.

This is my first project using GIT and it has been a huge savior. Learning how to save project states helped me practice coding incrementally with a focused goal. Reverting broken projects to working states was a huge help and time saver. I will certainly continue to use GIT for future projects.

If I were to redo this project I would certainly spend more time preparing my code. I originally jumped in with a rough metal idea of what I wanted the final result to be. I learned I should have documenting my goals and work towards a better structure before getting much code out.

**Bibliography**

Khronos Group. “GLUT - The OpenGL Utility Toolkit.” OpenGL.org, [www.opengl.org/resources/libraries/glut/](http://www.opengl.org/resources/libraries/glut/).

Khronos Group, “OpenGL 2.1 Reference Pages.” <https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/>.

University of Canterbury, Christchurch, New Zealand. Mukundan, R. (Spring 2017). *COSC 363*: *Computer Graphics.* Lecture.