COMP 2501

FINAL GAME PROJECT

ZOOM ZOOM GAME

BY

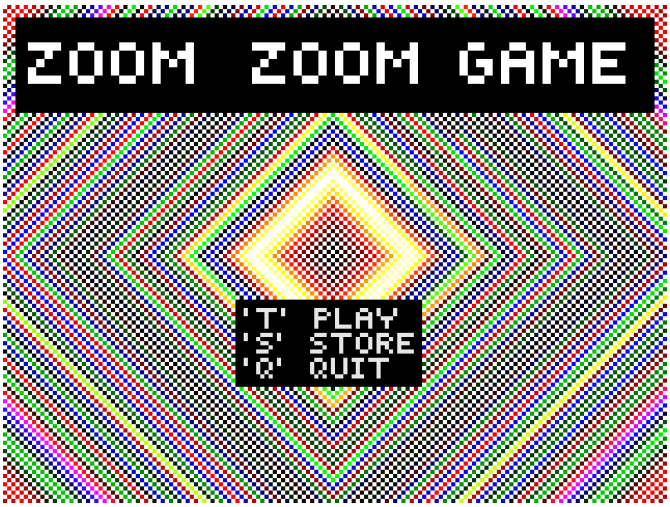
JEREMY CRINSON - 101008617

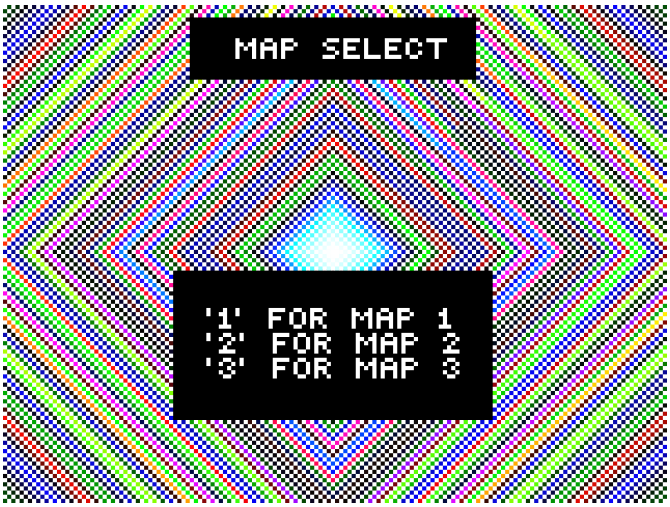
THEOFILOS PLESSAS - 100981341

Our game “ZOOM ZOOM Game” is a combination of both a survival game, and a racing game in which the players goal is to both escape the police following you around the track, and complete 3-5 laps (course dependant) while confined to 1 of 3 tracks chosen as the game starts. Scattered throughout the various track are coin tiles. When the player goes over any of these coin tiles, the player is awarded a coin, equal to 100 points. Using these points, the player will be able to purchase any of 3 different upgrades to give them more of an advantage in the races. Once the player completes a lap of the course they will be awarded 100 points (1 coin’s worth). As well as this another enemy will spawn that will chase them, just as the other enemies did. This enemy, however, is much faster then the default ones that spawn at the beginning of the race and thus the player will have to be more wary of them.

How to play:

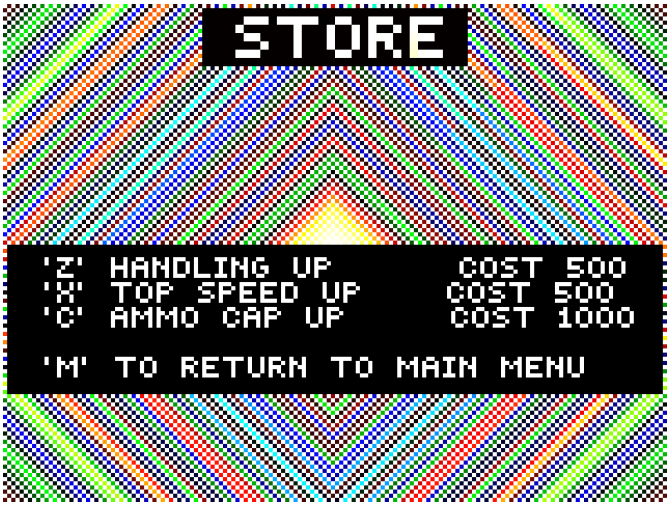
When the game launches, the player will be greeted by our Main Menu screen. The player will be given 3 keys that act as a navigation tool. Pressing ‘T’ will send the player to the Map Select Menu. In this menu the player has the option to choose 1 of 3 maps, each of which is unique. The player can make the selection by pressing ‘1’ , ‘2’ or ‘3’ and sending themselves into the map of their choice. When in game, the player can control the car using standard WASD controls. W will accelerate the car up to a maximum, A and D will rotate the car left and right respectively, allowing the player to turn corners or face enemies.





The S key will decelerate the car to a stop, and if held, will start to reverse the car. In game the SPACE key is used to fire off one round of the player's ammo in the direction the car is currently facing, allowing for the destruction of enemy vehicles. Each piece of ammo can take out multiple enemies, however there is a wait time the player must take in between shots. At any point in game, the player can press the 'M' key to return them to the main menu. However, if the player chooses to do this, they will lose all points they have earn currently that round, meaning that the player can only collect points he has earned that round by finishing the race.

If the player has collected enough money in game, they can start to buy items in the Store. To navigate to the store, press 'S' from the main menu screen. When the player enters the Store screen they will see a familiar menu to those of the Main Menu and Map select screens there will be key associated to the purchase of different upgrades the player can choose which will upgrade an aspect of their car. Each upgrade will also have a cost associated to each of them indicating the amount of money the player will need to earn in game to be able to purchase. The players main goal is to survive and collect money to then buy upgrades. Please note, there is a maximum amount of both handling and top speed cap. This is to prevent the player from having a car that is "too upgraded", which is humanly impossible to control.



Technical requirements:

The following will be the list of technical requirements from the “Prototype revisited” post on CULearn, each with an explanation on how we did or did not achieve a certain goal.

- Physics-based movement, collisions
    - This requirement is mostly completed, however improvements can be made. As for the movement, the player’s, enemy’s, and bullet’s movement are all physics based in terms of Euler integration. The part that could potentially used some improvement is in regards to the collisions. We do have collision detection and reaction, albeit for some objects. For example, when the player shoots a bullet at the enemies, both the enemies that the bullet has collided with and the bullet detect each other and respond accordingly. Due to the way we designed the game, this simply means eliminating the enemy and deleting the bullet. The player checks collisions with the track, such as running over a coin tile or running into the grass, and earns points or slows down accordingly. Player also earns “heat” when in contact with one or multiple enemies. However, we could not get dynamic collision resolution to work, such as enemies bouncing off other enemies and the player when a collision is detected, resulting in an unsatisfactory “layering” effect. However, for all practical purposes of our game, this bouncing effect isn’t necessarily needed for the game to function.
    - Please note, the police cars are supposed to be able to drive at regular speed over grass. They aren’t as fast as the player (in most cases) and their “off-road tires” is purposefully added in to add difficulty to the game.
- Movement through transformations
    - Movement done through transformations is fully implemented in a neat and efficient way. Originally we planned on making the world focus on the

player's movements so it would always stay in the centre, and rotate the world based on the players rotation so the player would always remain upright, and the world would change orientation. After implementing this, though, we felt it was better to just implement movement based on the player, as including orientation made the game much more difficult and unsatisfactory to play.
    - On top of this, any object in the game world that is outside of the screen's dimensions based on the player's position is not rendered but still updated, making the game run significantly better on PC's that have an older or less efficient graphics card.

- Hierarchical transformations used in some way
    - Currently the player is at the top of the hierarchy and the game world and its entities are transformed to adjust to the players virtual position in the game world.
- Particle system effect (explosion, sparks,...)
    - Currently in our game, we have not implemented a "true" particle system effect as explained on assignment 5, since this was something we planned on adding in at the end, but school caught up and then we had a group member leave.
    - That being said, in both the winter map (map 1) and the desert map (map 2), we do include a "faux" particle system that mimics snow/ a sandstorm on the map. While this particle system is technically a static clear image being rolled over and over again, both this image and the HUD's movement are not based off the player's movement, and use a completely different "static" rendering system to display.
- Finite state machine for AI
    - Again, due to time limitations our AI is directed to simply follow the player, rather than have multiple different states.

- Multiple game states (eg.,store)
    - We do have multiple game states implemented fully. These states are:
        - Main Menu
        - Store
        - Game
        - Settings
- Multiple game entities with different behaviours
    - We do have various different game entities that do different things. For starters, we obviously have the player, who is a human controller object.
    - We also have the various enemy/police objects, who all move on their own and are considered different types that the player in game. These enemies come in 2 different forms, including a squad car and a S.W.A.T. vans
    - We have bullets, which act exactly as the name suggests and are only created, updated, and displayed when shot. These objects travel in a straight line and collide with enemies to eliminate them
- Multiple terrain types with different effects
    - We have various different kinds of background or terrain objects. These include:
        - Regular track pieces, including straights and turns
        - Edge pieces, which slow down the player
        - Coin tiles, which give the player points
        - Race and checkpoint tiles (checkpoints are visually the same as straight pieces) which keep track of where the player is in the map and how many laps he's completed
    - We also have 2 different map types for the 3 different maps
        - Map 1 includes a snowy biome with a hail/snow effect that constantly runs on the players screen, as well as snow on the side rather than grass.

        - Map 2 contains a sandy biome, with a sideways windy sandstorm particle effect that runs across the players screen and sand as edge boundaries that slow the player.
        - Map 3 is just a standard race track, featuring grass boundaries that also slow down the player and no particle effects.
- Game world larger than screen
    - Our game world is larger than the screen and stimulates the players movement by sliding underneath the players car model. This affects all game world objects including bullets and enemies.
    - As mentioned earlier, the render function for actually displaying both the background and the game objects actually only draws what is in the player's field of vision (the screen) but keeps updating all the game objects in the world via their positions.

Gameplay:

A simple description of a typical gameplay session would be to choose your desired map, go around the track a few times collecting money and avoiding the enemies, including the ones spawning each lap, until you are eventually caught/destroyed and kicked back to the main menu where you can choose to start another round on a different map with a different environment, or to go to the store to spend the money earned during the last match to purchase upgrades for the players car.

Additional Features:

Some additional things we added to the game include reading the map from a text file and populating the world matrix with different tiles. This allows us to simply edit the text file to change the track instead of having 3 tracks simply hardcoded into a class or something. This is done be populating the text file in a table like arrangement of different numbers, these numbers are then read into the program and sent to an array to be stored. A second array is also being populated with different texture objects as the first array is being populated and this second array is what the player sees when driving around the world.

We also added sound to our game as an extra, We've added some music to the menus as well as some sound effects in game when shooting and when money is collected. This was done using mmsystems.h and the PlaySound() function and making a custom Sound class that takes advantage of these functions. Though it does have its limitations where no more than one sound might be played at one time but since its an extra feature we decided not to dwell on it for to long. There is also a mute key ('H') for the menu music.

Design Notes:

- General Design
    - The theme of the game we decided to go with is a more 2D, arcade style top down implementation. Because of this, we thought it was only appropriate to use more pixelated sprites in our implementation. This include all menus, game entities, and map sprites.
- Player
    - The player's design is pretty standard for a top down 2D racing game. The controls are the standard WASD, where 'W' will send the player forward in whichever direction he is facing, while 'A' and 'D' will change the direction of the player. 'S' will slow the player down and eventually allow the player to travel in reverse, just like a regular car. The 'Space' button just simply fires bullet.
    - The player also has a certain 'heat' value that he must retain during the course of the race. Once the player's heat value goes past 1500, the sprite will change to a more damaged version of the player, indicating that the player should probably get away from the police or shoot them soon. Once the heat value hits 3000, the player loses and the race is ended.
    - The player also has a minimum speed they need to be going in order to turn (including when in reverse) in order to make the car feel slightly more realistic and also add some extra difficult when the player ends up running into the side terrain.

- Enemy
    - Enemies are similar to the player in their control scheme, as it uses Euler integration and a target positioning system in order to go forward/backwards.
    - The enemies use a very basic AI in order to track the player. We would have liked to implement a slightly better AI, but as of now the enemy simply tracks the player and tries to “corner” him in, to increase the player’s heat.
    - Due to the speed and advantages given to the player, we decided to allow the police cars to travel off-road, unlike the player. This is included in order to be somewhat realistic, where the fast sports car would go much slower with slick tires on grass, while police cars are much tougher and more prepared to travel any terrain.
- Bullet
    - The bullet is very simple, like most bullets are. Bullets are used to eliminate enemies. Bullets are fired by pressing ‘Space’.
    - Bullets also have a reload time. The reload time is approximately 1 second, but this can make a significant difference when the player is stuck in the slower grass.
    - Bullets can eliminate more than one enemy. The reason we implemented this is for both speed and usability, since there is a reload time. In this way, bullets can travel through and eliminate multiple enemies at a time. This is also due to this being the players only weapon.

- Background/Map
    - We implemented the background/map in a way that it was tile based. This is used for 2 reasons. First of all, the style of our game is a more old school, pixelated style. So, square evenly sized tiles fits with the style of the game. Secondly, we implemented maps in a tiling system rather than a color detection/image map in order to make our lap system work. The way we implement laps is through a series of ‘checkpoint’ tiles. So, in order to complete a lap, the player must have gone through certain checkpoint tiles in a certain order in order to get the lap added when passing over the finish line. In this way, the player can’t go backwards around the map and still get the points
- Sound
    - Sound in our game, while not a visual object, is a very basic entity that plays certain sound effects.
    - On the Menu, a little tune plays in the background, just to give some noise to being on a simple menu. While there is not direct indication of this, this sound is mutable by pressing the key ‘H’, since after listening to the menu sound multiple times, it may get annoying.
- Menu objects
    - Due to the way we combined Model and View (see Implementation Notes), we created the game entity “Menu” as a way to implement objects that are seperate from the main game and are purely used as visuals. The reason behind this is due to the way the OpenGL library treats visual objects, and requires them to have a position on the screen based on their coordinates. For example, all menu visuals, the players HUD/points that are displayed in the rain, as

well as our particle effects all use this object as a base. In essence, they move on their own and not relative to the player's position.

Implementation Notes:

- comments on algorithms you used, especially things you invented
- Modified Model-View-Controller format: "(MV)C"
    - Our game features a modified Model-View-Controller format, as the title suggests. From when we started the project, we knew that we should have a group or vector of objects that are used in the game engine, a controller to control input, and the main/game loop that doesn't really do much other than set the game loop running and set up basic inits such as textures.
    - Since the implementation for most of the early demo code used the "GameEntity" class, which had both a render and update function, we based most of our code off of this and merged the View aspect into the Model/GameEntity class. For static objects that are essentially just displayed and don't interact with main gameplay, we created a separate class which include a different render function (see Menu above) that doesn't get pushed into our updateables vector.
    - Essentially, the ownership/hierarchy of or classes works as follows:
        - Main: owns Model*, Controller*
        - Controller: owns Model*
        - Model: owns all Game Entity objects except some menu/sound objects that are purely implemented in main.

- Text-Based Map
    - One algorithm that we implemented is our map system. Essentially, we wrote an algorithm that takes in certain *.txt files, reads these files upon map loading, and set sprites accordingly, rather than having a hard-coded in sprite sheet.
    - This implementation also makes the game much faster and allows for more customizability. This is due to the text files being essentially just an array of ints that give length and width, as well as value. This allows for many text files to be easily edited in a text editor, rather than sifting through many lines of code. The uses can even implement their own maps by writing simple map functions.
    - These map text based tile sheets also work very efficiently by saving in game CPU costs by loading the maps at the beginning of the game, rather than constantly reading off a sprite sheet or loading maps at runtime, making actual gameplay much smoother
- Durable/Bug resistant game loading
    - Within the model class, we have also implemented 2 seperate load game and unload game functions in order to make re-loading the game much more durable than say, deleting and creating a brand new model object for each new map load.
    - These functions also allow for the model class itself to save most of the players data for purchases in the shop.
- Easy on Older/Less Powerful GPUs
    - As mentioned earlier, the game objects (except for certain static menu/HUD objects) all move based on the players movements, thus giving a game map bigger than the screen. This, when just looping through all displayable graphics objects and printing them, whether or not

they are actually viewable from the player's screen, make the game run incredibly slowly on older computers. To solve this problem, all game entities are displayed if and only if they are on the player's screen. At the same time though, they are continuing to be updated as to not lose positions and properties, thus sharing the load between the GPU and the CPU.

Known Bugs and limitations:

When in a new instance of the game where the player has completed 0 laps and has not made any new enemies spawn, if the player lines up all current enemies and kills them with a single bullet 10% of the time the game will crash throwing a NULL pointer exception. We think this is caused by the bullet trying to destroy something that is not there anymore.

There is also the issue we've had attempting to have the enemies collide with each other and we think thats a limitation in the way we've designed enemies to behave.

Sound in our game also has some major limitations because of the Audio library we've chosen to use, making it so that only one audio file can be played at any one time (although this may be a misunderstanding on our part).

Certain key presses (in specific, the Store) have been known to call more than once (once per frame rather than once per press). This, especially when in the store or using a cheat code such as 'I' (gives more money) can result in the player buying more than 1 upgrade if the key is held for too long. We were planning on implementing a callback function in controller that fixes this bug, but due to our limited understanding of OpenGL and time limitations, we were unable to fully implement this.

In our game, we have certain cheat codes. For example, pressing 'I' will grant the player more money and pressing 'L' will instantly complete the race. However, as

mentioned above, holding down ‘I’ for too long will result in the player’s HUD of money not being displayed (money is > than 10,000).

Comments on individual contribution:

A lot of the base OpenGL code and simple game objects were provided by tutorials and assignments by the TAs and Dr. Mould.

- Theo
    - Game-states
        - Menu
        - Store
            - Player upgrades
                - Turning buff
                - Max speed
        - Map select
    - Sound
    - Bullet -> enemy detection
    - Prototype write-ups / progress report
    - Animation cycle
        - Police flashing lights
    - Ammo capacity
    - “Reload” timing / firing delay
- Jeremy
    - Game Engine
        - Model-View and Controller Architecture (see above)
        - Model’s update function and game loop
        - Initiation and deletion of objects from the game
        - Efficient displaying of game objects on screen
    - Background/Map

        - Reading map from *txt file (see above)
        - Displaying map background
        - Creating checkpoint/lap system
        - Coin tile for points
        - Snow/Grass/Sand tiles slow down player
        - Loading new “advanced” enemy upon completing a lap
    - Transformation-Based Movement (OpenGL matrix modifications)
        - Objects display and move based on player’s movement
        - staticRender() function for objects not based on players movement
    - Player/Enemy/Bullet
        - Physics-Based Euler integration for player, enemy, bullet
        - Player ‘Heat’ when in contact with police
        - Basic police AI
        - Player’s HUD and point system
        - Bullet physics and firing
    - Static Rendering/particle effects
        - Sandstorm/Snowstorm effects
- Collaboratively
    - Upgrades for Player/Store integration
    - All Sprite Creation
        - Drawing and Creation
        - Implementing them into the game
    - Editing final drafts of write ups
    - 2501 Final Submission write-up (this document)
    - Debugging
        - When one of us had a problem, we would collaboratively debug on Discord to brainstorm the problem.

Post-Mortem:

During our time in this project, we had a few issues. There were 3 main issues we had with this project. This first of these issues was with OpenGL itself. Now that we understand the library and can partially work and read it, there are few problems we can't fix if given the right amount of time. However, towards the start of the course, we both felt that the skeleton code was extremely hard to understand. While it was went over partially in class, or at least sudo code, we both feel that if we had went over more-so on how the actual skeleton code worked, and how to use it, rather than theoretically understand what it does and the end result, it would have been much more beneficial to our project, since we could get started on bigger and more foundational tasks such as writing the game engine (MVC) earlier, and build from there

The second major issue that we both personally had was scheduling. With the size of this project and what it all entails, we found that keeping a consistent schedule for the project itself, completing assignments, as well as maintaining good grades in our other 4 classes, the projected schedule was very hard to maintain. Especially when simultaneously taking classes that also had big game design projects such as COMP 3004.

The last issue we had was with a group member who has dropped the course. The trouble with this was that he dropped the course right when major features were due, specifically the final project and this 15 page write-up. While if the member had dropped the course earlier, it would have been possible to potentially find another member to join our group and share the workload more evenly. But since the member that dropped the course did so at such a late period in time, the rest of us had to complete a significantly greater proportion of work than expected, meaning that it was much more difficult to implement features that we had planned on implementing.

Screenshots: