# *CRIME-LAPSE:*

## *ROBO RANGER RELOADED*



## Abstract

*A Sci-fi game built using the BabylonJS game engine. For physics, the Cannon.js physics engine is used while sound effects are handled by the Ion.sound library.*



Cameron Smith

Kanishk Singh

# TABLE OF CONTENTS

# Introduction

## Controls

| | |
|---|---|
| WASD | = move |
| Left Click | = shoot |
| Right Click | = blink/teleport |
| Space | = jump (and double jump) |
| E | = go back in time (currently no real use for this but it's still cool) |

## Game Summary

Currently the game has 3 rounds. For the first round, the objective is to kill all of the mole bots. These mole bots pop in and out of the ground. When they pop out of the ground they shoot a laser and spin. After a certain amount of time, they will go back underground and pick a random spot to reappear at. Once all of the mole bots are killed, the player moves on to the second round where they have to kill spiderbots (A.K.A. ala-akbots if that's not too offensive). These bots simply chase the player and try to explode in your face. Finally, once the spiderbots are all killed, the player moves on to the final round, the "Simon Boss Fight". This boss is resembles a bullet hell boss that requires the player to shoot his buttons that match the color of the floor. If the player shoots the wrong button, he ejects 4 bombs. If the player shoots the correct button, he loses 5% of his health and then give the floor a new random color (that is never the same as the current color). At 75%, 50%, and 25% health, Simon enters enrage mode. When Simon is enraged he has a special attack pattern which is much harder to deal with, and does not take any damage. During this time, the player has to simply focus on survival until the enrage mode is finished. Once Simon is defeated, the game is over.

# Problems / Unfinished Goals

We've ran into a TON of problems while making this game, but fortunately most of them got solved. The biggest unsolved problem right now is the fact that the spiderbots don't have materials and are not cell shaded like the rest of the scene. This was not the case when we just had one spiderbot for testing, as the problem lies in the cloning process. We were able to get one spiderbot working perfectly, but when we clone it, it suddenly ignores the multi-material and just uses the first submaterial in the list. On top of this, when trying to give it a cell shaded material, it suddenly ruins the animation and makes it walk really weird.

Another slight issue is that sometimes the player can teleport through the walls and therefore fall outside of the map. We've found that it is hard teleport through the walls when we are trying to, but when we are actually playing the game normally it happens. A solution could be thickening the wall meshes, or checking if the player is out of bounds every frame and then moving him back to his previous position (since position is stored every few frames for going back in time anyways, this wouldn't be too bad). But anyway, for the time being if you accidentally teleport through a wall you'll just have to press E to go back in time.

# Workflow

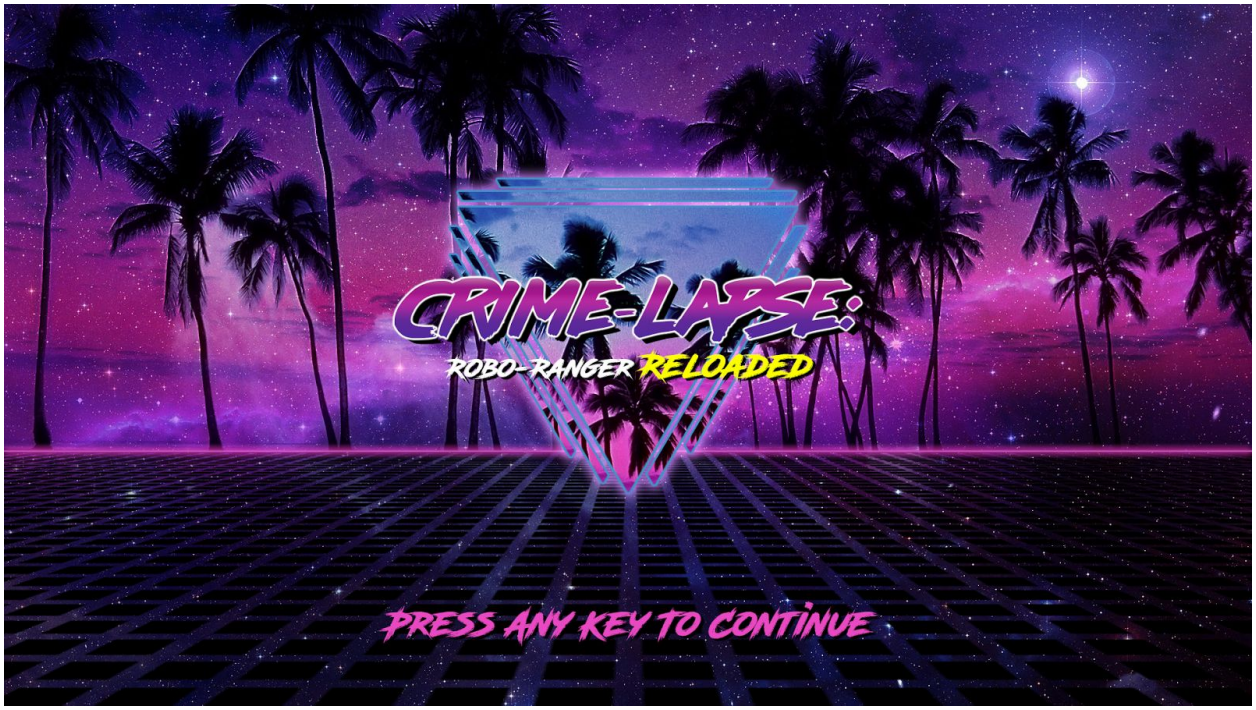
## init.js

This script is in charge of initializing global variables and creating the necessary game components. This includes setting up the babylon game engine and scene, and creating the canvases and cameras.

## titlescreen.js (Currently Disabled)

This script simply draws the starting screen. A keydown event listener is put in place to call the startLoading() function once any key is pressed. This event listener is then removed once the event triggers.

For some unknown reason, this causes problems on certain computers. So right before submitting this project it was removed (startLoading is called immediately instead of waiting for user input). This is a bit depressing because we spent so much time making the title screen super fresh, here's what it looked like.



# util.js

This script contains various utility functions that are used across multiple scripts. These functions include:

RGBtoHSV(color):

converts a BABYLON.Color3 to [H, S, V].

HSVtoRGB(color):

converts [H, S, V] to BABYLON.Color3.

rotateVecAboutAxis(vec, theta, axis):

rotates a vector about an given axis by given theta.

giveCellShaderMaterial(mesh):

Takes a mesh and converts its material (including submaterials) to a cell shaded material. Due to the way Babylon handles shader materials, they require a texture and won't work with just a diffuse color alone. This is a problem since many of the meshes being loaded simply use a standard material with a diffuse color and no texture. This problem was solved by drawing the mesh's old material's diffuse color to a 1x1 pixel canvas and then converting that canvas to a dataURL which is then used as the new materials texture image.

changeShaderMatColor(mesh, color):

Using a method very similar to giveCellShaderMaterial, this function takes both a mesh and a BABYLON.Color3 and then converts the color of the mesh's cell shaded material's texture to the specified color.

# loadscreen.js

This script makes use of an array called meshLoadOrder. The elements in this array contain attributes describing the meshes that will be loaded, and these meshes will be loaded in the same order as they are indexed within the array. The attributes of each element are as follows:

| | |
|---|---|
| filename: | url to the .babylon mesh file |
| scene: | scene to add the mesh to |
| onsuccess: | function that is called upon successful import |

When startLoading() is called, the first element of meshLoadOrder is popped off the queue and it's attributes are passed as parameters in the BABYLON.SceneLoader.ImportMesh() function. The function loadCallback() is also passed as the progressCallBack parameter. This function receives a loading event which contains the progress of the current mesh being loaded. When the progress reaches 100%, it recursively calls the load function on the next mesh in the queue until there are no meshes left. When no meshes are left in the queue, the function startGameLoop() is called.

# gameloop.js

This script handles everything that goes on in the game during each frame. To make things more organized, many of the processes are separated into separate functions as listed below:

| | |
|---|---|
| keyListener(): | checks to see which keys are currently pressed |
| moveBullets(): | iterates through bullet list, moves and deletes when necessary. |
| playerHandler(): | does everything related to player, including handling timers for various player actions. |
| enemyHandler(): | calls each enemy's update() function and disposes of them when they are dead. |
| drawUI(): | draws HUD / UI on a separate canvas. |
| scene.render(): | renders the scene. |

The function startGameLoop() initiates this game loop. To ensure that it does not start until the scene is completely initialized, the render loop is also wrapped inside a scene.executeWhenReady() call.

## input.js

This script holds the main keyListener function that is called each frame. A keydown event listener is put in place to set the value of keys[e.keyCode] to true whenever a key is pressed. Thus to see if a key is pressed you simply have to check if keys[keyCode] is true. If WASD is pressed, the player's mesh is moved. If left click is pressed, it either makes a pointerlock request (to lock the mouse in the screen) or shoots if pointerlock is already enabled. Right click initiates player's teleport/blink, which disables player movement for a few frames while he moves in the appropriate direction. This direction is set based on a raycast that is shot from the player, which is then rotated based which WASD buttons are pressed. The length of this teleport is also determined by this raycast, if the raycast hits a wall that is only 5 units from the player, the teleport will only move 5 units. Even with this in place, it sometimes glitches, causing the player to fly through the wall.

## UI.js

This script holds the drawUI function that is called each frame which draws the UI/HUD on a separate overlay canvas. The elements of the HUD are pretty straight forward, such as health, number of enemies left, FPS, etc.

# LIBRARIES

## BabylonJS and Cannon.js Physics

We use BabylonJS as our main game engine. In order for Babylon's physics to work, it needs to make use of either oimo.js or cannon.js physics engines. To set up the physics, we simply call scene.enablePhysics() after cannon.js has been included in the document.

We use BABYLON.physicsImpostors for collision detection. PhysicsImpostors can be given a mass, restitution (bounciness), and collision filter masks/groups among many other properties. To make an object static, it simply needs to be given a mass of 0 and it will not be affected by gravity or collision. Here's an example of setting up the floor mesh's physicsImpostor:

```
floor.physicsImpostor = new BABYLON.PhysicsImpostor(floorMesh,
BABYLON.PhysicsImpostor.BoxImpostor, {
        mass: 0,
        restitution: 0,
        nativeOptions: {
                collisionFilterMask: PLAYER_MASK + ENEMY_MASK,
                collisionFilterGroup: GROUND_MASK
        }
}, scene);
```

The collisionFilterMask tells the physics engine which collision groups this mesh should collide with, while the collisionFilterGroup tells the physics engine which collision group this mesh belongs to. In this case, the floor only collides with enemies and the player. These masks also must be a power of 2 to work appropriately.

To setup collisions the registerOnPhysicsCollide function is called which takes another physicsImpostor to be collided with and a function that is called upon collision with this physicsImpostor.

Here is an example that resets the player's jump count when he collides with the floor's mesh:

```
floor.physicsImpostor.registerOnPhysicsCollide(player.mesh.physicsImpostor,
        function() {
                player.jumpCount = 2;
                player.isFalling = false;
        }
);
```

## Ion.Sound

This API makes playing sounds very easy. To load sounds, you simply have to call ion.sound with a parameter object that encapsulates the names of every sound you wish to load along with special properties for each. Examples of these properties are volume, preload, and multiplay. Another useful property is an ended_callback function, which is called when a sound has finished playing. This makes handling events a lot easier, an example where this is used is when the enemies begin spawning right after a dialogue finishes playing.

# ENEMY OBJECTS

## Enemy.js

Simple script that holds a few things for the enemies, such as the enemyList array, a variable keeping track of the number of enemies left, and the healthBarSpriteManager that is in charge of creating health bar sprites for all enemies.

## Molebot.js

This enemy's mechanics are driven by states. The state constants are as follows:

DOWN = 0, GOING_UP = 1, UP = 2, GOING_DOWN = 3

Each state has its own duration stored in the MoleBot's stateLengths array. Indexing is done according to the state constants i.e.
stateLengths[DOWN] = amount of time that bot will stay underground

Each frame, the bots stateTimer is decremented by deltaTime. When it reaches 0, the bot moves on to the next state. Since the constants were set in the appropriate order, this is done by simply incrementing the bots current state (using modulus to wrap back to 0)

```
If (stateTimer <= 0) {
        state = (state + 1) mod 4
        stateTimer = stateLengths[state]
}
```

Now that the states are appropriately timed and switched to, the bot behaves according to its current state simply using if statements  i.e.

```
If state == UP
        Spin
If state == DOWN
        Do nothing
If state == GOING_DOWN
        Move down
If state == GOING_UP
        Move up
```

When the bot first reaches the DOWN state, it selects a new random position within the boundaries of the room, causing it to reappear in a different spot each time it comes up.

## Spiderbot.js

There's quite a few problems with the spiderbots currently. Due to issues with cloning of the original spiderbot mesh, there's a large possibility of their walk animations getting messed up. On top of this, only one of them has the true material, while the others are all white.

When spiderbots have been activated, every frame the distance and direction to the player is calculated. The direction is normalized and then multiplied by the spiderbot's move speed and finally applied to the spiderbots position to make him move towards the player. If the distance is less than the spiderbots distance threshold, the spiderbot's "withinRangeCounter" is incremented. When this counter reaches a set number, the spiderbot will begin detonating. This counter is put in place to prevent the spiderbot from detonating the second the player is in range, and making it only detonate if it is within range for a set number of frames.

When detonating, the detonateTimer is decremented. Once the detonateTimer reaches 0, it creates an explosion at the spiderbots position and removes the spiderbot. Similiarly, if the players shoot raycast hits the spiderbot's bounding box, the detonating boolean is set to true and detonateTimer is set to 0 to cause it to explode the next frame.

# Simon.js

Simon has an array of button meshes, each one having a color associated with it along with a function that is called when it gets shot by the player. When a button is shot, it's associated color is compared with the color of the floor. If the color does not match, it triggers simon to shoot out a bomb in each direction with a random force (simply create four enemyBomb objects and position/rotate them appropriately). If the color matches, then it subtracts 5% from Simons health. If this causes simons health to reach 75%, 50%, or 25%, it enables Simon's "enraged" property, which causes him to switch to his enraged attack pattern, which lasts for a time set by his enrageTimer.

As of right now, Simon has two different attack patterns. The attackPatterns store these properties:

| | |
|---|---|
| name: | name of the attack pattern |
| fireRate: | number of shots per second |
| fireRateAccel: | acceleration of fireRate |
| bulletAccel: | initial bullet speed acceleration |
| individualBulletAccel: | acceleration given to each bullet |
| bulletSpeedMin / Max | min/max bullet speed to be wrapped around |
| rotateAccel: | rotation acceleration of simon mesh |
| rotateSpeedMin / Max | min/max rotate speed to be wrapped around |

After the attack patterns have been saved in Simon's attackPatterns array, they can easily be switched by calling Simon.switchToAttackPattern(attackPattern) which sets all of simons properties equal to the attack pattern's properties.

Simon starts off underground, and does not move above ground until the "simon_intro_1" sound has finished playing (this sound is triggered after killing off all of the spiderbots). Once he makes it all the way above the ground (y = 0) it plays the "simon_intro_2" sound where he introduces himself, and later instructs the player to shoot his red button. If his red button is shot while he is deactivated, it plays the "simon_intro_3" sound. Once this sound is finished playing, he is activated and therefore begins updating each loop.

# Game Objects

## Player.js

hurt(damage):

Subtracts damage from players health and plays a sound

goBackInTime() / traverseTime():

Player has two array storing previous positions and camera rotations. Every few frames, the player's current position and camera rotation is pushed to this array, and the oldest position/rotation is removed.

When goBackInTime is called, it starts playing a sound effect and sets the player's goingBack property to true. While this property is true, the player can no longer move. Instead, the player's traverseTime function is called each frame.

When traverseTime is called, it sets the players position and the camera's rotation equal to the values being popped off of their associated arrays. Once the array's length reaches 0, the player's goingBack property is set back to false and the player can then again move. goBackInTime will only work when the array of stored positions/rotations is full (60). Since the positions/rotations are only stored every few frames (5 to be exact) the effect of going back in time goes 5 times as fast as the actual time it took to fill up the array, leading to a cool effect.

## Bullet.js

The bullets in this game are actually 2d sprites rather than meshes. However they do have invisible meshes that are used not only for their physicsImpostors (allows collisions) but also for making their movement a bit easier.

EnemyBullet

- has a sprite and invisible mesh
- collision with player hurts the player and the bullet gets disposed
- collision with wall simply disposes the bullet

EnemyBomb

- very similar to EnemyBullet in that it makes use of a sprite and invisible mesh, except the sprite is different and the mesh's physics impostor is given a mass (so it is affected by gravity).
- collision with player/wall/floor creates a new Explosion object at the bomb's positions and kills the bomb.

moveBullets()

called each frame, iterates through the bullet list backwards, moving each bullet and disposing/removing from list if they are dead.

# Gun.js

Originally we planned on having multiple guns, so this should really be a variable rather than class. Also, originally the gun had ammo, but creating a nice looking reload animation was a bit tricky, so we decided to for an overheat mechanic. After playing with the overheat mechanic for a while, it got pretty annoying so now the gun simply lights up red as the player shoots, but never actually overheats.

Whenever the player shoots, it knocks the gun's local z position back 5 units. Then in each frame, the gun is moved back towards its natural position by $1/5^{th}$ difference.

Rather than using a projectile, we use raycasting along with a 2d line that is drawn from the tip of the gun to the raycast hit position. Whenever this raycast hits a mesh, it checks to see if mesh.hurt exists. If it exists, it calls mesh.hurt() which reduces health from the mesh's parent enemy object.

# Explosion.js

Explosions make use of the BABYLON.SpriteManager for animating their sprite. An explosion is given a position and a radius. When it is first created, its distance from the player is calculated, and if this distance is less than radius, it hurts the player for 10% of his max health.

# Resource Citations

Music

Orbital Colossus: https://opengameart.org/content/space-boss-battle-theme
n-Dimensions: https://opengameart.org/content/space-dimensions-8bitretro-version
Xeon: https://opengameart.org/content/xeon-theme-remastered

Models

FPS Arm: http://www.blendswap.com/blends/view/69621
Bomb Sprite: https://opengameart.org/content/bomb-sprite-vector-image
Gun Attachments: https://opengameart.org/content/lowpoly-gun-attachments
Spiderbot: https://opengameart.org/content/spiderbot-10
Ship: https://opengameart.org/content/hextraction-base-player-pod
Skybox: https://opengameart.org/content/space-skyboxes-0
Gun: https://opengameart.org/content/smg-2

SFX

All Dialogue: http://www.oddcast.com/home/demos/tts/tts_example.php
Laser Sound: https://opengameart.org/content/sci-fi-laser-fire-sfx
Electric Sounds: https://opengameart.org/content/electric-sound-effects-library
Explosion: https://opengameart.org/content/big-explosion
Hurt Sound: https://opengameart.org/content/15-vocal-male-strainhurtpainjump-sounds
Robotic Arm Sound: http://www.youtube.com/watch?v=896eFuYsPjM

Created by Us

Simon model
Space Arena model
Gun model was heavily modified
Molebot model