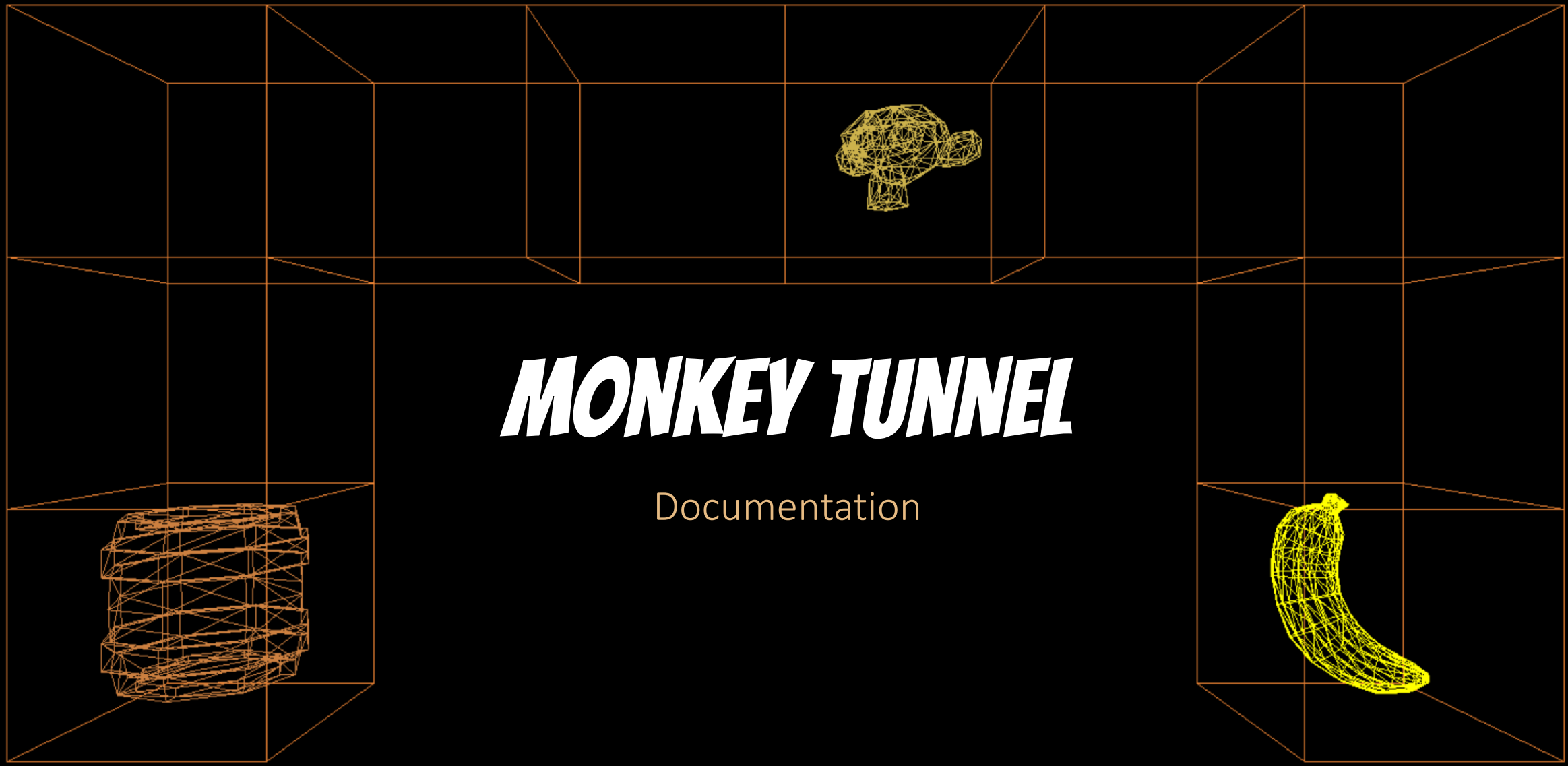


MONKEY TUNNEL

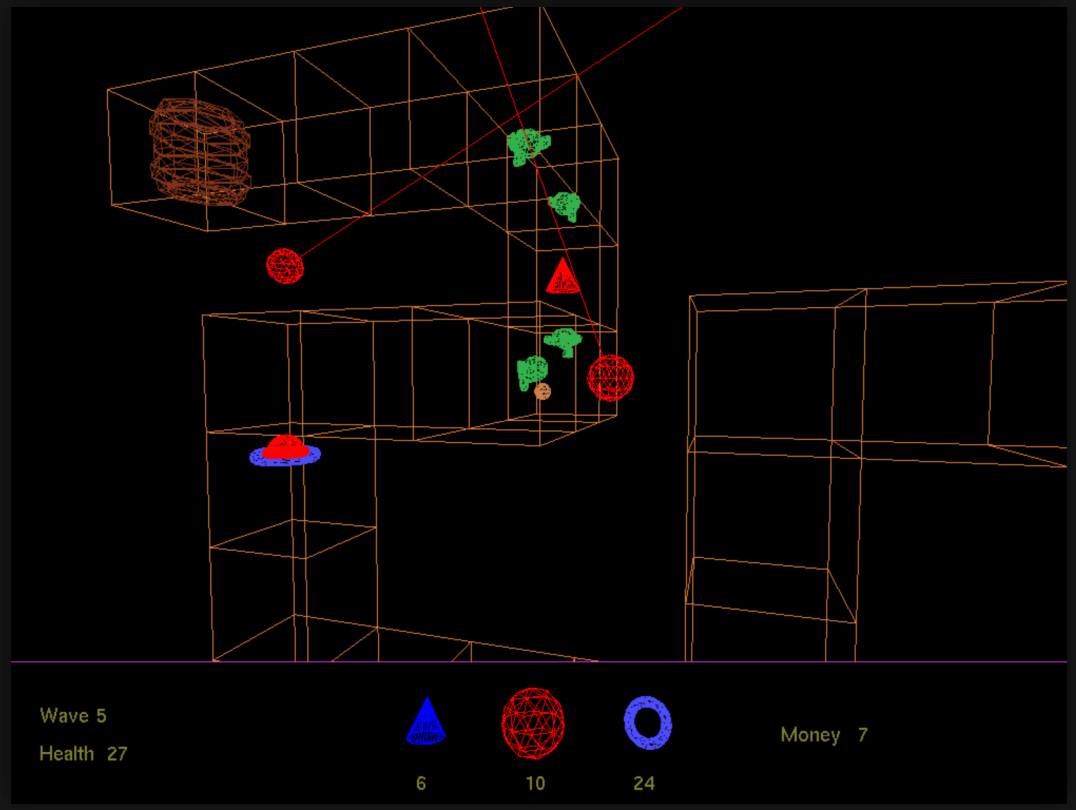
Documentation

Overview

Monkey Tunnel is a Tower defense game set in (3D) space!

Oh no! A barrel of monkeys has appeared in your star system.

Take control of your ship to buy defenses and protect the banana!



How To Play

Purchase and place towers to defend the tunnel.

You can't place towers inside the tunnel, or on other towers, shown when the ship is red. Saws follow the player's ship and will damage monkeys.

The colour of the ship can also be used to better determine your position in the 3D space.

Survive 10 waves to win !

GTK

Game Tool Kit (gtk) is a static lib I created to help prepare for the challenge.

Game.h contains the base game class used to connect with the NEXT API.

A “game” manages a list of scenes, and facilitates moving between scenes.

First scene added is used on startup.

gtk/Game.h

```
namespace gtk {

    class Scene;

    class Game {

    public:

        // Deletes all scenes in the scene map
        virtual ~Game();

        // Starts the active scene
        void Start();

        // Updates the active scene
        void Update(const float& deltaTime);

        // Calls Draw on active scene's renderers
        void Render(const float& width, const float& height);

        // Deletes all scenes
        void Shutdown();

        // Shutdown active, change active, start active
        void SwitchScene(std::string key);


    protected:

        // Override in custom game class
        virtual void Init() = 0;

        // Adds a new scene to the scene map
        void AddScene(const std::string& key, Scene* newScene);

    protected:

        // Scenes
        Scene* m_ActiveScene;
        std::unordered_map<std::string, Scene*> m_SceneMap;

    };
}
```

Game/Games.h

```
class TowerDefenseGame : public gtk::Game
{

public:

    void Init() override
    {
        // Add the scenes of the game here
        AddScene("MainMenu", new MainMenu(*this));
        AddScene("L1", new TD_Level_1(*this));
        AddScene("L2", new TD_Level_2(*this));
    }
};
```

Provided GameTest.cpp

```
//--------------------------------------------------------------------------------
// Data
//--------------------------------------------------------------------------------

TowerDefenseGame game;

//--------------------------------------------------------------------------------

//--------------------------------------------------------------------------------
// Called before first update. Do any initial setup here.
//--------------------------------------------------------------------------------
void Init()
{
    // Creates scenes and starts the first scene
    game.Init();
    game.Start();
}

//--------------------------------------------------------------------------------
// Update your simulation here. deltaTime is the elapsed time since the last update in ms.
// This will be called at no greater frequency than the value of APP_MAX_FRAME_RATE
//--------------------------------------------------------------------------------
void Update(float deltaTime)
{
    // Updates all active components in the active scene

    game.Update(deltaTime/1000);

    float f = GLUT_WINDOW_WIDTH;
    float s = GLUT_WINDOW_HEIGHT;
}

//--------------------------------------------------------------------------------
// Add your display calls here (DrawLine,Print, DrawSprite.)
// See App.h
//--------------------------------------------------------------------------------
void Render()
{
    // Calls draw in active renderers in active scene
    game.Render(1024.0f, 768.0f);
}
```

Scenes

Scenes provide an interface for creating and managing entities, behaviors, renderers, and collision.

Each frame, the scene calls update on all behaviors, updates the scene graph, checks collision, and calls draw on renderers.

UpdateGroups and RenderLayers are used to schedule update and draw calls.

Only colliders in the same CollisionGroup will be checked against each other.

The scene allocates memory for all entities on Init, and deletes all managed memory on Shutdown.

Game/Scenes.h

```
class SceneTemplate : public gtk::Scene
{

public:

    SceneTemplate(gtk::Game& game) : gtk::Scene(game) {}

protected:

    // Called by game when scene starts
    void Setup() override
    {
        using namespace gtk;

        UpdateGroup group = CreateUpdateGroup();
        RenderLayer layer = CreateRenderLayer();
        CollisionGroup colGroup = CreateCollisionGroup();

        Entity& camera = CreateEntity();
            AddCamera(camera, new PerspectiveCam(1, 100, 80));

        Entity& parent = CreateEntity();
            AddBehavior(parent, group, new BehaviorTemplate());
            AddRenderer(parent, layer, new RendTemplate());

        Entity& child = CreateEntity(parent);
            AddBehavior(child, group, new BehaviorTemplate());
            AddRenderer(child, layer, new RendTemplate());
            AddCollider(child, colGroup, new SphereCollider());

    }

    // Called after all entities are updated
    void PostUpdate() override
    {

    }
};
```

Basic scene setup

SceneObjects

Entities, Renderers, Colliders, Cameras, and ObjectPools all inherit from SceneObject.

SceneObject provides useful functions used when “scripting”.

SceneObject communicates with the scene using the entity ID to facilitate data flow to all an entity’s related objects (behaviors, renderers, etc.).

gtk/SceneObjects.h

```
11
12  ////////////////////////////////////////
13  //             SceneObject            //
14  ////////////////////////////////////////
15
16  // Base class for anything mananged by the scene
17  // Provides handy functions
18
19  class Entity;
20  class Scene;
21
22  class SceneObject
23  {
24      friend class Scene;
25
26  public:
27
28      SceneObject();
29      virtual ~SceneObject() {}
30
31      const std::string& GetName();
32
33      const vec3& Pos();
34      const vec3& Pos(const vec3& pos);
35      const vec3& Pos(const vec3& pos, const bool& add);
36
37      const vec3& Rot();
38      const vec3& Rot(const vec3& pos);
39      const vec3& Rot(const vec3& pos, const bool& add);
40
41      const vec3& Scale();
42      const vec3& Scale(const vec3& pos);
43      const vec3& Scale(const vec3& pos, const bool& add);
44
45      const unsigned int& ID() const;
46
47      Entity& Parent();
48      const mat4& TRS();
49      const vec3& GlobalPos();
50
51      vec3 Right();
52      vec3 Up();
53      vec3 Forward();
54
55
56      const int& State();
57      const int& State(const int& newState);
58
59      void Trigger(const int& code);
60      void TriggerCollision(Entity& other);
61      void SetColor(const vec3& color);
```

Entity

In GTK an Entity contains the transform data and linked list connections to its parent and children.

Entities can only be created using a Scene class.

Entity& child = CreateEntity(parent);

Disabling an entity will disable all of its behaviors, renderers, and colliders, then does the same for its children.

gtk/SceneObjects.h

```
271  ////////////////////////////////////////
272  //              Entities              //
273  ////////////////////////////////////////
274
275  class Entity : public SceneObject
276  {
277      friend class Scene;
278      friend class SceneObject;
279      friend class Camera;
280
281  public:
282
283      const mat4& GetTRS();
284
285      const bool& Active(const bool& setActive);
286      const bool& Active();
287
288  private:
289
290      // Call CreateEntity() from a Scene
291      Entity();
292      Entity(const Entity&) = delete;
293
294      void AddChild(Entity* child);
295
296      mat4 CalcTRS();
297      mat4 CalcTR();
298
299      void UpdateRootTRS();
300      void UpdateTRS();
301
302  private:
303
304      std::string _Name;
305
306      Entity* _Parent;
307
308      std::vector<Entity*> _Children;
309
310      bool _Active; // Update comps and draw
311
```

Behavior

A Behavior is where game logic is created.

Calling Trigger from an Entity reference will “trigger” all of its the behaviors using the provided code.

I would love to use a scripting language for these.

Game/Components.h

```
79  class BehaviorTemplate : public gtk::Behavior
80  {
81
82  public:
83      BehaviorTemplate() {}
84
85      // Called before the first update
86      void Start() override {}
87
88      void Update(const float& deltaTime) override {}
89
90      // Use for sending messages through entities
91      int Trigger(const int& code) override
92      {
93          return 0;
94      }
95
96      void OnCollision(Entity& other) override {}
97
98  };
```

Renderers

Renderers are intended to connect with the provided NEXT API drawing functions.

For example, OBJRenderer loads the vertex data from a file during Scene setup, then uses the App::DrawLine function to draw the models.

Currently, each OBJRenderer opens and reads the file. My backlog includes adding Model objects for the renderer to reduce load time.

Game/Renderers.h

```
179  class OBJRenderer : public gtk::Renderer
180  {
181
182  public:
183
184      OBJRenderer(std::string filePath, gtk::vec3 color = vec3(0.5f, 0.5f, 0.2f))
185          : _vbo(), _ibo(), Renderer(color)
186      {
187          LoadObject(filePath);
188      }
189
190      void Start() override
191      {
192          // Called first frame
193      }
194
195      void Draw() override
196      {
197          gtk::vec4 s;
198          gtk::vec4 e;
199
200          // Functions from SceneObject
201          gtk::mat4 model = TRS();
202          gtk::mat4 view = GetView();
203          gtk::mat4 proj = GetProj();
204
205          gtk::mat4 mvp = proj * view * model;
206
207          // Draw Model
208          for (int i = 0; i < _ibo.size(); i += 2)
209          {
210              s = { _vbo[_ibo[i]].x, _vbo[_ibo[i]].y, _vbo[_ibo[i]].z, 1 };
211              e = { _vbo[_ibo[i + 1]].x, _vbo[_ibo[i + 1]].y, _vbo[_ibo[i + 1]].z, 1 };
212
213              s = mvp * s;
214              e = mvp * e;
215
216              // Both points infront of camera
217              if (s.z > 0 && e.z > 0)
218              {
219                  App::DrawLine(
220                      s.x / s.z, s.y / s.z,
221                      e.x / e.z, e.y / e.z,
222                      _color.x, _color.y, _color.z);
223              }
224
```

Object Pools

Object Pools provide an interface for creating and reusing many of the same object.

An object pool is created in Scene::Setup and can be used in behaviors to spawn objects.

I use them for Bullets, Shooters, Lasers, and Monkeys.

```
39
40      void GeneratePool() override
41      {
42          for (int i = 0; i < _count; i++)
43          {
44              // Create entity and add it to pool
45              Entity* entity = &_scene.CreateEntity("bullet");
46              _pool.push(entity);
47
48              // Setup here
49              _scene.AddBehavior(*entity, _group, new BulletB());
50              _scene.AddRenderer(*entity, _layer, new OBJRenderer(".\\TestData\\ico.obj", vec3(0.8f, 0.5, 0.3f)));
51              _scene.AddCollider(*entity, _colGroup, new SphereCollider(2.0f));
52              entity->Scale(0.5f);
53
54              entity->Active(false);
55          }
56
57      }
```

BulletPool in Game/ObjectPools.h

Monkey Path Creation

I wanted to make an efficient way to create paths, so I could easily create a few levels without hardcoding the positions.

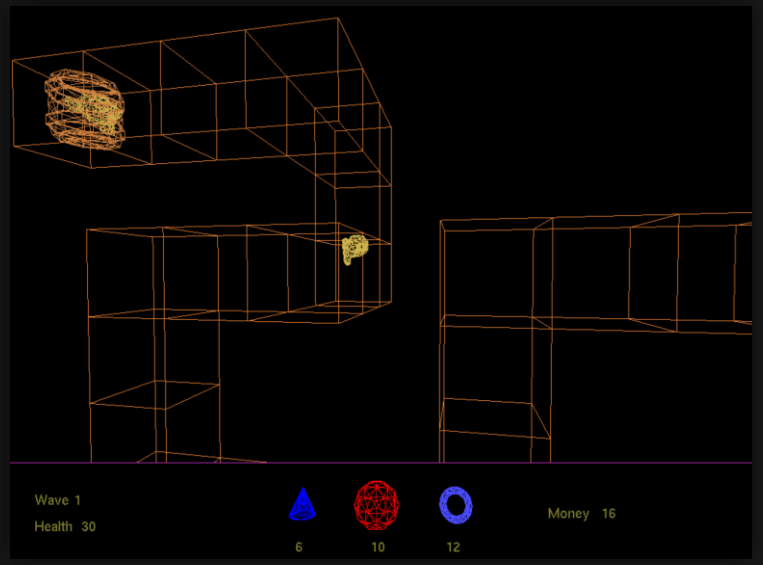
The path vector is also used by the monkeys to interpolate between each node.

```cpp
166
167  void CreatePath(const std::vector<vec3>& nodePos, RenderLayer layer, CollisionGroup colGroup, const vec3& color)
168  {
169      int i = 0;
170      for (const vec3& pos : nodePos)
171      {
172          Entity& node = CreateEntity("PathNode" + std::to_string(i));
173          AddRenderer(node, layer, new CubeRenderer(color));
174          AddCollider(node, colGroup, new SphereCollider());
175          node.Scale(3.0f);
176          node.Pos(pos);
177          i++;
178      }
179  }
180  void AddToPath(std::vector<vec3>& nodes, Direction dir, int count)
181  {
182      ASSERT(nodes.size() > 0);
183
184      switch (dir)
185      {
186      case LEFT :
187
188          for (int i = 0; i < count; i++)
189          {
190              nodes.push_back(vec3(nodes.back()) + vec3(-6.0f, 0.0f, 0.0f));
191          }
```

Game/Scenes.h

```cpp
366          //// PATH ////
367  #pragma region PathCreation
368
369          std::vector<vec3> path;
370
371          // Start Node
372          path.push_back(vec3(-28.0f, 12.0f, 0.0f));
373
374          AddToPath(path, RIGHT, 3);
375          AddToPath(path, FORWARD, 2);
376          AddToPath(path, DOWN, 2);
377          AddToPath(path, BACK, 2);
378          AddToPath(path, LEFT, 2);
379          AddToPath(path, DOWN, 3);
380          AddToPath(path, RIGHT, 4);
381          AddToPath(path, UP, 3);
382          AddToPath(path, RIGHT, 3);
383          AddToPath(path, BACK, 1);
384          AddToPath(path, UP, 3);
385
386          CreatePath(path, layer1, cursorSelectionCol, RED);
387
388  #pragma endregion
389
```

Game/Scenes.h



Monkey Waves

Creating the waves of monkey spawns uses a similar approach to path creation. A Wave object contains a list of SpawnGroups.

A SpawnGroup contains a monkey type, how many should be spawned, and the delay between each spawn.

A Waves vector is created in the Scene Setup then used by the BarrelOfMonkeysB behaviour (in Game/Components.h) to time the spawns and trigger monkey types correctly.

BarrelOfMonkeysB also controls the level's win and lose conditions.

Game/Components.h

```
581  struct Wave
582  {
583      int numMonkeys = 0;
584      std::vector<SpawnGroup> _spawnList;
585
586      // Count, Type of monkey, spawn delay between each
587      void AddToWave(const int& count, const int& monkeyType, const float& delay)
588      {
589          numMonkeys += count;
590          _spawnList.push_back({ count, monkeyType, delay });
591      }
592
593  };
594
```

Game/Scenes.h

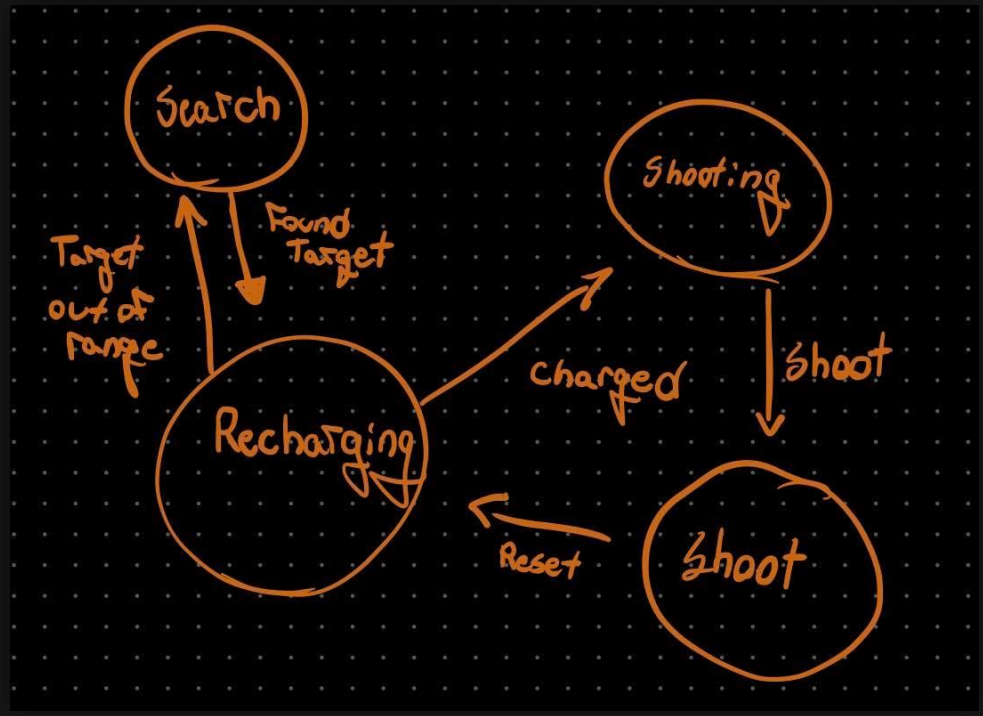
```
443          // Waves //
444
445          std::vector<Wave> waves;
446
447          Wave wave1;
448          wave1.AddToWave(6, STANDARD, 3.0f);
449          waves.push_back(wave1);
450
451          Wave wave2;
452          wave2.AddToWave(6, TINY, 1.0f);
453          wave2.AddToWave(3, BRUTE, 3.0f);
454          waves.push_back(wave2);
455
456          Wave wave3;
457          wave3.AddToWave(1, BOSS, 3.0f);
458          waves.push_back(wave3);
459
```

Shooter and Laser Towers

Shooter and Laser towers use the towerSight collision group to search for a target using a sphere collider.

Then using the targets position they fire either a laser or bullet at the target using a separate bullet collision group.

Shooter bullets can only hit one target, while the Laser can hit multiple using Ray-Sphere collision.



Basic Laser states

Player’s Ship and Saws

The player’s ship, called Cursor in the code, manages the player’s health, money, and tower purchasing in the CursorB behavior (Game/Components.h)

Saws are child objects of the player and are activated as they’re purchased. Saws follow the player and will damage monkeys on contact.