BOID SIMULATION

UNIVERSITY OF BUCHAREST

COMPUTER GRAPHICS 2D PROJECT

PĂCURARIU RĂZVAN MIHAI LUPARU IOAN TEODOR

2025-2026

CONTENTS

CONCEPT

The scope of this project was to implement a boid simulation, alongside some other features, in an easily extensible way.

USED TRANSFORMATIONS

This project uses all the standard glm transformations a 2D OpenGL project might need. The ortho transformation is used to properly map the entities' coordinates onto screen space, alongside the translate, rotate and scale transformations, which are used both for the camera and the entities to change their position, orientation and zoom/size in the 2D space. UnProject is also used for mapping mouse coordinates back into the scene space.

DESCRIPTION AND ORIGINALITY

The project includes, alongside the already mentioned standard boid simulation, nests. On initialization, each boid is assigned a random nest to which is attracted to, as in its velocity vector is guided towards the center of the nest, alongside the usual components that are added to it by a boid simulation. When a boid gets too close to a nest, it will try to orbit it (and on certain simulation parameters, it will also achieve this stably).

At a constant interval, a random boid is chosen for their nest and the nests of all its nearby neighbors to be reassigned to the same random one. This encourages more dynamic movement that displays the swarming effect of the boids more clearly.

There are clouds that pan from left to right and loop back around to accentuate the feeling of a top-down perspective of some birds. These clouds move at different speeds, based on a randomly chosen height value, and also have random a random scale and rotation, giving the illusion of a more diverse selection of cloud shapes.

As for controls, there is the ability to zoom in or out using the scroll wheel, and also to track either a random boid by pressing T or a random nest by pressing Y (pressing T/Y again will return the camera back to the standard view). There are also three sliders at the bottom of the screen, which control three of the main parameters of the boid simulation, encouraging the user to interact with the application for longer. These sliders are controlled using the `[`, `]` keys, `;`. `'` keys and `,`, `.` keys respectively. The shaders are rendered with a custom (pair of) shader(s), giving them a border, and also allowing the slider bar length to be changed using a uniform. By clicking the left mouse button, boids will now be attracted to your mouse cursor, allowing you to vaguely guide them; this mode can be disabled by clicking the left mouse button again.

On the backend side of things, a mini engine was implemented to allow for easier work with OpenGL, there being a Renderer which can render and update a Scene which stores a list of Entities. Each Entity has a Mesh, which stores the VAO, VBO, and EBO for the entity, and it can also choose which shader to use. There is also a TextureManager that manages textures and an InputManager that allows for easy bindings between an input method and a lambda. Other classes present in the engine are the Camera(representing the camera in a scene, stores position, rotation, zoom, other helper functions and members (such as for tracking)), Texture(interface over the usual way to create a texture in OpenGL, easy to interact with), Shader(class abstraction over loadShader.cpp), App(class representing the application itself, handles properly initializing the Window and Renderer and so on, also the actual scene content and input bindings are specified in it), Window(handles creating the GLUT window and also stores some info about it).

This engine was designed to be extensible, as can be seen by the boid simulation being represented by a BoidScene, class derived from Scene, and the Boid, Nest, Cloud and Slider classes, derived from the Entity class. For the derived entities in the boid scene in particular, they use shared resources such that all boids use the same Mesh and Texture data, similarly for nests and clouds, to save on memory usage. The clouds use 4 different textures which are chosen at random upon initialization.

All the textures present in this application were hand made by the team members and were not taken from an asset site.

IMAGES

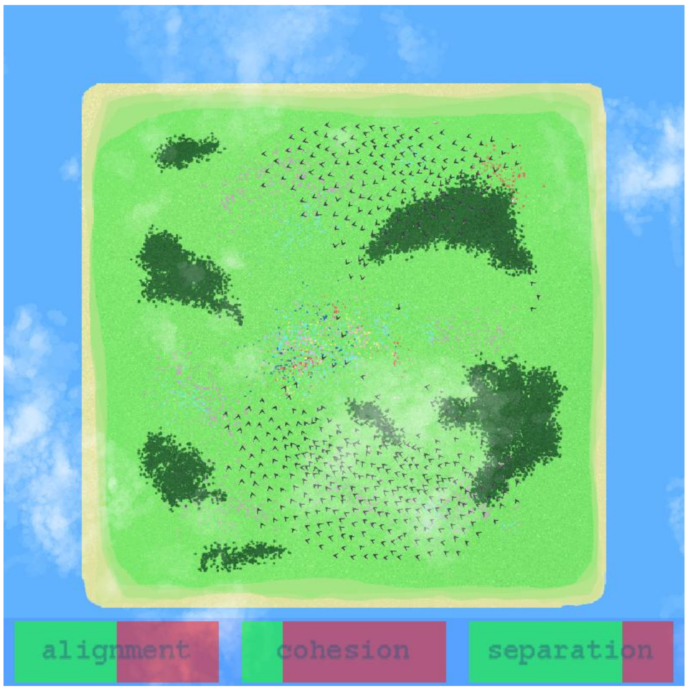

Figure 1: The main view of the application. In it can be seen the boids moving as expected, nests and clouds, alongside the 3 sliders at the bottom.

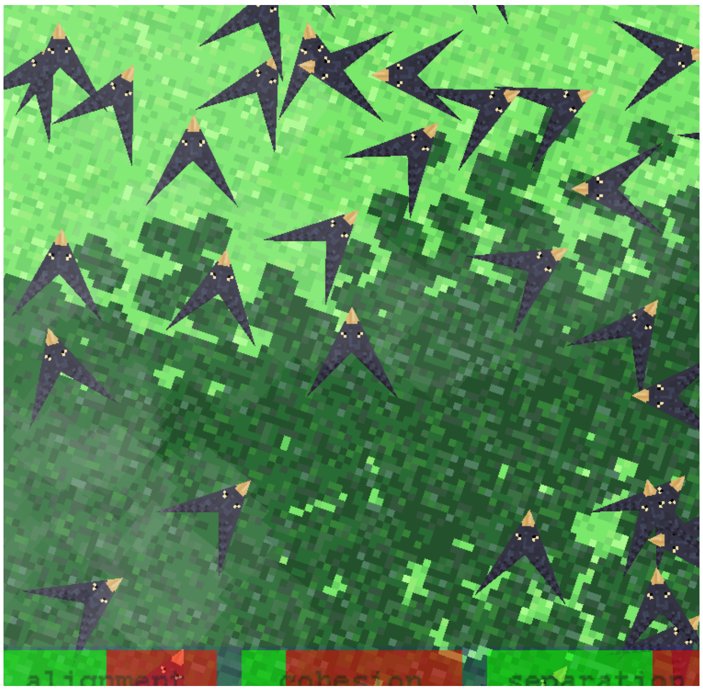

Figure 2: Cropped and zoomed in view of a boid being tracked.

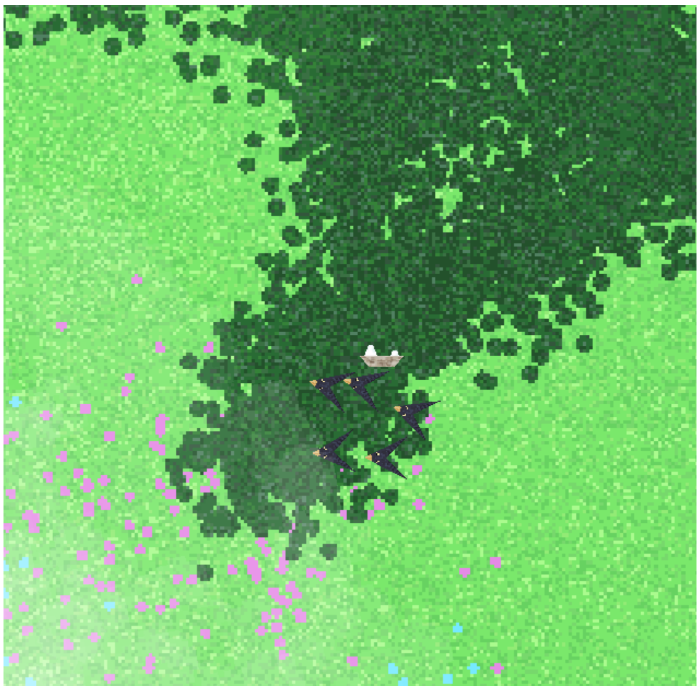

Figure 3: Cropped and zoomed in view of a nest. Boids can be seen circling it.

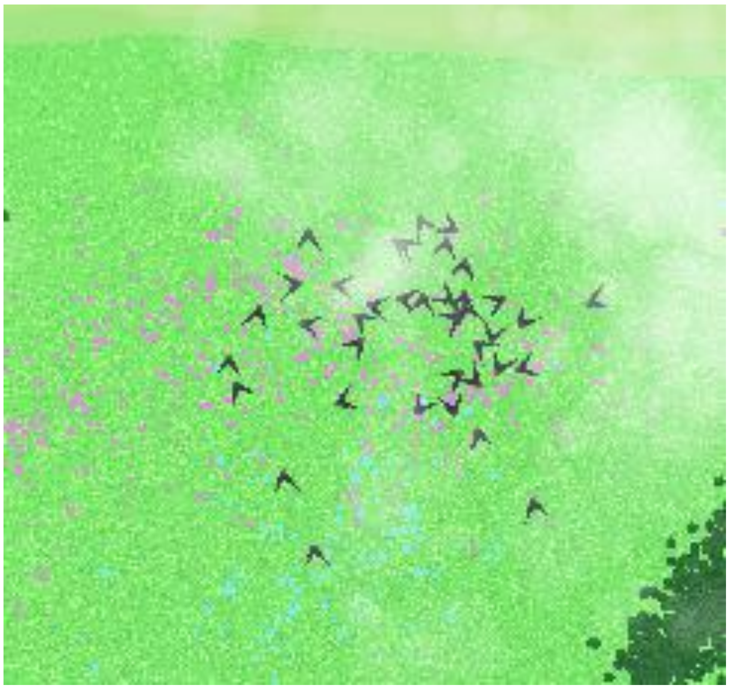

Figure 4: Boids being attracted to the mouse cursor (notice the lack of a nest).

INDIVIDUAL CONTRIBUTIONS

Păcurariu Răzvan Mihai implemented the engine, the initial boid simulation and sliders.

Luparu Ioan Teodor implemented the nests, clouds, mouse cursor attraction, and also helped with coming up with the ideas for this project.

Both worked a roughly equal amount on the textures.

GITHUB

The repo for this project can be found at https://github.com/razvanpacku/ComputerGraphics-Project-2D.

REFERENCES

- This site was used as a reference for implementing the boids algorithm.
- The usual sources of help for fixing problems or figuring out how to do something (StackOverflow, LLMs, VS's autocomplete and copilot).

APPENDIX: SOURCE CODE

main.cpp

```
#include "src/Engine/App.h"

int main(int argc, char* argv[])
{
	int32_t argc0 = static_cast<int32_t>(argc);
	App::Init(argc0, argv);
	App* app = &App::Get("Boids");

	app->Run();
}
```

App.h

```
#pragma once
#include <string>

#include "Window.h"
#include "Renderer/Renderer.h"

#define WINDOW_WIDTH 800
#define WINDOW_HEIGHT 800

//forward declaration
class Entity;

class App
{
public:
	static void Init(int32_t argc, char* argv[]);
	static App& Get(const std::string& name = "", uint16_t width = WINDOW_WIDTH, uint16_t height =
WINDOW_HEIGHT);

	void Update();

	void SetEntityTracking(bool boidOrNest = false);

	void Run();
private:
	App(const std::string& name, uint16_t width, uint16_t height);
	~App();

	static int32_t argc;
	static char** argv;

	float lastFrameTime = 0.0f;
	float deltaTime = 0.0f;

	bool isTrackingEntity = false;
	Entity* trackedEntity = nullptr;
	void UpdateEntityTracking(Entity* entity);

	Window *window;
	Renderer *renderer;
};
```

App.cpp

```
#include "./App.h"

#include "Renderer/TextureManager.h"
#include "InputManager.h"
#include "../Boid/Boid.h"
#include "../Boid/BoidScene.h"

#include <cmath>
#include <ctime>

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

int32_t App::argc = 0;
char** App::argv = nullptr;

App::App(const std::string& name, uint16_t width, uint16_t height)
{
	srand(static_cast<unsigned int>(time(0)));

	window = new Window(width, height, name, argc, argv);
	renderer = new Renderer(this);

	// Set up camera
	Camera* camera = new Camera();
	renderer->SetCamera(camera);
	//camera controls
	InputManager::RegisterMouseWheelAction([camera](int direction) {
		camera->ProcessMouseScroll(direction);
		});
	InputManager::RegisterKeyAction('t', [this]() {
		this->SetEntityTracking(false);
		});
	InputManager::RegisterKeyAction('y', [this]() {
		this->SetEntityTracking(true);
		});
	InputManager::RegisterMouseButtonAction([this](int button, int state, int x, int y) {
		if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
		{
			// Convert screen (x,y) to world coordinates
			Camera* cam = renderer->GetCamera();
			if (!cam) return;

			// Flip Y
			float winX = static_cast<float>(x);
			float winY = static_cast<float>(window->GetHeight() - y);

			glm::vec4 viewport(0.0f, 0.0f, static_cast<float>(window->GetWidth()),
static_cast<float>(window->GetHeight()));
```

```
                              // glm::unProject expects model (modelview) and proj matrices.
                              // Pass view matrix as the "model" param
                              glm::vec3 winPos(winX, winY, 0.0f);

                              glm::vec3 worldPos = glm::unProject(winPos, cam->GetViewMatrix(), cam-
>GetProjectionMatrix(), viewport);

                              // Toggle mouse attract in the BoidScene
                              Scene* sc = renderer->GetScene();
                              if (sc)
                              {
                                    BoidScene* bs = dynamic_cast<BoidScene*>(sc);
                                    if (bs)
                                    {
                                          bs->ToggleMouseAttractAt(glm::vec2(worldPos.x, worldPos.y));
                                    }
                              }
                  }
                  });

      // Update attract point continuously while mouse-attract mode is active
      InputManager::RegisterCursorPosAction([this](int x, int y) {
                  Camera* cam = renderer->GetCamera();
                  if (!cam) return;

                  float winX = static_cast<float>(x);
                  float winY = static_cast<float>(window->GetHeight() - y);
                  glm::vec4 viewport(0.0f, 0.0f, static_cast<float>(window->GetWidth()),
static_cast<float>(window->GetHeight()));
                  glm::vec3 winPos(winX, winY, 0.0f);
                  glm::vec3 worldPos = glm::unProject(winPos, cam->GetViewMatrix(), cam-
>GetProjectionMatrix(), viewport);

                  Scene* sc = renderer->GetScene();
                  if (sc)
                  {
                              BoidScene* bs = dynamic_cast<BoidScene*>(sc);
                              if (bs && bs->IsMouseAttractActive())
                              {
                                    bs->SetMouseAttractPos(glm::vec2(worldPos.x, worldPos.y));
                              }
                  }
                  });

      //Load textures
      auto tex1 = TextureManager::Load("textures/dev.png");
      auto nestTex = TextureManager::Load("textures/nest.png");
      auto boidTex = TextureManager::Load("textures/boid.png");
      auto backgroundTex = TextureManager::Load("textures/background.png");
      TextureManager::Load("textures/alignment.png");
      TextureManager::Load("textures/cohesion.png");
      TextureManager::Load("textures/separation.png");
```

```
	// Define a square mesh
	std::vector<glm::vec2> vertices = { {-0.5f,-0.5f}, {0.5f,-0.5f}, {0.5f,0.5f}, {-0.5f,0.5f} };
	std::vector<glm::vec3> colors = { {1,1,1}, {1,1,1}, {1,1,1}, {1,1,1} };
	std::vector<glm::vec2> texCoords = { {0,0}, {1,0}, {1,1}, {0,1} };
	std::vector<uint32_t> indices = { 0,1,2, 2,3,0 };
	//
	auto squareMesh = renderer->AddMesh("square", std::make_shared<Mesh>(vertices, colors,
texCoords, indices));

	Scene* scene = new BoidScene(renderer);

	auto nestMesh = renderer->AddMesh("nest", BoidScene::CreateNestMesh());
	Nest::InitSharedResources(nestMesh, nestTex);
	auto boidMesh = renderer->AddMesh("boid", BoidScene::CreateBoidMesh());
	Boid::InitSharedResources(boidMesh, boidTex);

	std::vector<std::shared_ptr<Texture>> cloudTextures = {
		TextureManager::Load("textures/cloud1.png"),
		TextureManager::Load("textures/cloud2.png"),
		TextureManager::Load("textures/cloud3.png"),
		TextureManager::Load("textures/cloud4.png"),
	};
	Cloud::InitSharedResources(squareMesh, cloudTextures);

	dynamic_cast<BoidScene*>(scene)->AddBackground(squareMesh, backgroundTex);
	dynamic_cast<BoidScene*>(scene)->InitNests(5);
	dynamic_cast<BoidScene*>(scene)->InitBoids(500);
	dynamic_cast<BoidScene*>(scene)->InitClouds(20);
	dynamic_cast<BoidScene*>(scene)->AddControlEntities(squareMesh);

	renderer->SetScene(scene);
}

App::~App()
{
	delete window;
	delete renderer;
}

void App::Init(int32_t argc, char* argv[])
{
	App::argc = argc;
	App::argv = argv;
}

App& App::Get(const std::string& name, uint16_t width, uint16_t height)
{
	static App* instance = nullptr;
	if (instance == nullptr)
	{
		instance = new App(name, width, height);
	}
	return *instance;
}

void App::SetEntityTracking(bool boidOrNest)
{
	isTrackingEntity = !isTrackingEntity;

	//special behaviour for boidscene
	if (isTrackingEntity){
		if(boidOrNest){
			trackedEntity = dynamic_cast<BoidScene*>(renderer->GetScene())-
>GetRandomNest();
		}
		else{
			trackedEntity = dynamic_cast<BoidScene*>(renderer->GetScene())-
>GetRandomBoid();
		}
		renderer->GetCamera()->targetEntity = trackedEntity;
	}

	if (isTrackingEntity && trackedEntity) {
		renderer->GetCamera()->hasTarget = true;
	}
	else {
		renderer->GetCamera()->hasTarget = false;
	}
}

void App::UpdateEntityTracking(Entity* entity)
{
	trackedEntity = entity;
	renderer->GetCamera()->targetEntity = entity;
}

void App::Run()
{
	glutMainLoop();
}

void App::Update()
{
	float currentFrameTime = static_cast<float>(glutGet(GLUT_ELAPSED_TIME)) / 1000.0f;
	deltaTime = currentFrameTime - lastFrameTime;
	lastFrameTime = currentFrameTime;

	if (renderer && renderer->GetScene())
		renderer->GetScene()->Update(deltaTime);
}
```

Window.h

```
#pragma once
#include <GL/glew.h>
#include <GL/freeglut.h>

#include <cstdint>
#include <string>

class Window
{
public:
	Window(uint16_t width, uint16_t height, const std::string& name, int32_t argc, char** argv);
	~Window();

	uint16_t GetWidth() const { return width; }
	uint16_t GetHeight() const { return height; }
private:
	uint16_t width, height;

};
```

Window.cpp

```
#include "./Window.h"

#include <iostream>

Window::Window(uint16_t width, uint16_t height, const std::string& name, int32_t argc, char** argv)
	: width(width), height(height)
{
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
	glutInitWindowPosition(100, 100);
	glutInitWindowSize(width, height);
	glutCreateWindow(name.c_str());
	GLenum res = glewInit();
	if (res != GLEW_OK) {
		std::cerr << "Error: " << glewGetErrorString(res) << std::endl;
		exit(1);
	}

	static Window* self = this;

	//stop window resizing
	glutReshapeFunc([](int w, int h) {
		glutReshapeWindow(self->width, self->height);
		});
}

Window::~Window()
{
}
```

InputManager.h

```
#pragma once
#include <GL/freeglut.h>

#include <functional>
#include <unordered_map>
#include <vector>

// static class for managing input
class InputManager
{
public:
	using KeyCallback = std::function<void()>;

	// Register a callback for a key press
	static void RegisterKeyAction(unsigned char key, const KeyCallback& callback);

	// Register mouse wheel callback
	static void RegisterMouseWheelAction(const std::function<void(int direction)>& callback);

	// Register mouse button callback: (button, state, x, y)
	static void RegisterMouseButtonAction(const std::function<void(int button, int state, int x, int y)>&
callback);

	// Register cursor position callback: (x, y)
	static void RegisterCursorPosAction(const std::function<void(int x, int y)>& callback);

	// Called by GLUT callbacks
	static void OnKeyPress(unsigned char key);
	static void OnMouseWheel(int direction);
	static void OnMouseButton(int button, int state, int x, int y);
	static void OnCursorPos(int x, int y);
private:
	static std::unordered_map<unsigned char, std::vector<KeyCallback>> keyCallbacks;
	static std::vector<std::function<void(int direction)>> mouseWheelCallbacks;
	static std::vector<std::function<void(int button, int state, int x, int y)>> mouseButtonCallbacks;
	static std::vector<std::function<void(int x, int y)>> cursorPosCallbacks;
};
```

InputManager.cpp

```
#include "./InputManager.h"

std::unordered_map<unsigned char, std::vector<InputManager::KeyCallback>> InputManager::keyCallbacks;
std::vector<std::function<void(int direction)>> InputManager::mouseWheelCallbacks;
std::vector<std::function<void(int button, int state, int x, int y)>> InputManager::mouseButtonCallbacks;
std::vector<std::function<void(int x, int y)>> InputManager::cursorPosCallbacks;

void InputManager::RegisterKeyAction(unsigned char key, const KeyCallback& callback)
{
	keyCallbacks[key].push_back(callback);
}

void InputManager::RegisterMouseWheelAction(const std::function<void(int direction)>& callback)
{
	mouseWheelCallbacks.push_back(callback);
}

void InputManager::RegisterMouseButtonAction(const std::function<void(int button, int state, int x, int y)>&
callback)
{
	mouseButtonCallbacks.push_back(callback);
}

void InputManager::RegisterCursorPosAction(const std::function<void(int x, int y)>& callback)
{
	cursorPosCallbacks.push_back(callback);
}

void InputManager::OnKeyPress(unsigned char key)
{
	auto it = keyCallbacks.find(key);
	if (it != keyCallbacks.end())
	{
		for (const auto& callback : it->second)
		{
			callback();
		}
	}
}

void InputManager::OnMouseWheel(int direction)
{
	for (const auto& callback : mouseWheelCallbacks)
	{
		callback(direction);
	}
}

void InputManager::OnMouseButton(int button, int state, int x, int y)
{
	for (const auto& callback : mouseButtonCallbacks)
	{
		callback(button, state, x, y);
	}
}

void InputManager::OnCursorPos(int x, int y)
{
	for (const auto& callback : cursorPosCallbacks)
	{
		callback(x, y);
	}
}
```

Entity.h

```
#pragma once
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include "Renderer/Mesh.h"
#include "Renderer/Texture.h"
#include "Renderer/Shader.h"

#include <memory>


class Entity
{
public:
	Entity(std::shared_ptr<Mesh> mesh, const glm::vec2& position = glm::vec2(0.0f), const glm::vec2&
scale = glm::vec2(1.0f), float rotation = 0.0f, std::shared_ptr<Texture> texture = nullptr,
std::shared_ptr<Shader> shader = nullptr,
		bool isGUI = false);
	virtual ~Entity();

	glm::vec2 position;
	float rotation;
	glm::vec2 scale;

	float opacity = 1.0f;

	bool useTexture = true;
	bool isGUI = false;

	void SetTexture(std::shared_ptr<Texture> tex) { texture = tex; }
	void SetShader(std::shared_ptr<Shader> sh) { shader = sh; }

	virtual void ApplyUniforms(Shader& shader) const;

	std::shared_ptr<Shader> GetShader() const { return shader; }

	glm::mat4 GetModelMatrix() const;
	void Draw() const;
private:
	std::shared_ptr<Mesh> mesh;
	std::shared_ptr<Texture> texture = nullptr;
	std::shared_ptr<Shader> shader = nullptr;
};
```

Entity.cpp

```
#include "./Entity.h"

Entity::Entity(std::shared_ptr<Mesh> mesh, const glm::vec2& position, const glm::vec2& scale, float rotation,
std::shared_ptr<Texture> texture, std::shared_ptr<Shader> shader, bool isGUI)
	: mesh(mesh), position(position), scale(scale), rotation(rotation), texture(texture), shader(shader),
isGUI(isGUI)
{
}

Entity::~Entity()
{
}

glm::mat4 Entity::GetModelMatrix() const
{
	glm::mat4 model = glm::mat4(1.0f);
	model = glm::translate(model, glm::vec3(position, 0.0f));
	model = glm::rotate(model, glm::radians(rotation), glm::vec3(0.0f, 0.0f, 1.0f));
	model = glm::scale(model, glm::vec3(scale, 1.0f));
	return model;
}

void Entity::ApplyUniforms(Shader& shader) const{
    shader.setMat4("model", GetModelMatrix());
    shader.setBool("useTexture", useTexture);
	shader.setFloat("opacity", opacity);
}

void Entity::Draw() const
{
    // Apply texture if available
    if (useTexture && texture)
    {
        texture->Bind();
        glEnable(GL_TEXTURE_2D);
    }
    else
    {
        glDisable(GL_TEXTURE_2D);
    }

	mesh->Draw();

    if (texture)
        glBindTexture(GL_TEXTURE_2D, 0);
}
```

Scene.h

```
#pragma once
#include <glm/glm.hpp>

#include <vector>
#include <memory>
#include <functional>

// Forward declaration
class Entity;
class Renderer;

class Scene
{
public:

    Scene(Renderer* renderer);
    virtual ~Scene();

    Entity* AddEntity(std::unique_ptr<Entity> entity);
    void RemoveEntity(std::unique_ptr<Entity> entity);

    void SetUpdateFunction(const std::function<void(float)>& func) {
        updateFunction = func;
    }
    void Update(float deltaTime);

    glm::vec3 backgroundColor = glm::vec3(0.75f, 1.0f, 1.0f);

    void Render() const;
protected:
    std::vector<std::unique_ptr<Entity>> entities;

    Renderer* renderer;

    std::function<void(float)> updateFunction;
private:
};
```

Scene.cpp

```
#include "./Scene.h"

#include "Renderer/Renderer.h"
#include "Entity.h"

Scene::Scene(Renderer* renderer) : renderer(renderer)
{
}

Scene::~Scene()
{
	entities.clear();
}

Entity* Scene::AddEntity(std::unique_ptr<Entity> entity)
{
	Entity* ptr = entity.get();
	entities.push_back(std::move(entity));
	return ptr;
}

void Scene::RemoveEntity(std::unique_ptr<Entity> entity)
{
	entities.erase(std::remove(entities.begin(), entities.end(), entity), entities.end());
}

void Scene::Update(float deltaTime)
{
	if (updateFunction)
		updateFunction(deltaTime);
}

void Scene::Render() const
{
	// First: render normal entities
	for (const auto& e : entities)
	{
		if (!e->isGUI)
			renderer->RenderEntity(*e);
	}

	// Then: render GUI entities
	for (const auto& e : entities)
	{
		if (e->isGUI)
			renderer->RenderEntity(*e);
	}
}
```

Camera.h

```
#pragma once
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

#define ORTHO_WIDTH 10.0f
#define ORTHO_HEIGHT 10.0f

//forward declaration
class Entity;

class Camera
{
public:
	Camera();
	~Camera();

	glm::vec2 position = glm::vec2(0.0f);
	float rotation = 0.0f;

	float zoom = 1.0f;

	bool hasTarget = false;
	Entity* targetEntity = nullptr;

	glm::mat4 GetViewMatrix() const;
	glm::mat4 GetProjectionMatrix() const;

	void ProcessMouseScroll(int direction);
private:
};
```

Camera.cpp

```
#include "./Camera.h"

#include "../Entity.h"

Camera::Camera()
{
}

Camera::~Camera()
{
}

glm::mat4 Camera::GetViewMatrix() const
{
	glm::mat4 transform;
	if (!hasTarget || !targetEntity) {
		transform = glm::translate(glm::mat4(1.0f), glm::vec3(position, 0.0f));
		transform = glm::rotate(transform, glm::radians(rotation), glm::vec3(0.0f, 0.0f, 1.0f));
		transform = glm::scale(transform, glm::vec3(zoom, zoom, 1.0f));
	}
	else {
		transform = glm::translate(glm::mat4(1.0f), glm::vec3(targetEntity->position, 0.0f));
		transform = glm::rotate(transform, glm::radians(targetEntity->rotation), glm::vec3(0.0f, 0.0f,
1.0f));
		transform = glm::scale(transform, glm::vec3(zoom, zoom, 1.0f));
	}
	return glm::inverse(transform);
}

glm::mat4 Camera::GetProjectionMatrix() const
{
	float halfWidth = (ORTHO_WIDTH / 2.0f) ;
	float halfHeight = (ORTHO_HEIGHT / 2.0f);
	return glm::ortho(-halfWidth, halfWidth, -halfHeight, halfHeight, -1.0f, 1.0f);
}

void Camera::ProcessMouseScroll(int direction)
{
	const float zoomStep = 0.1f;
	if (direction > 0) // scroll up
	{
		zoom -= zoomStep;
		if (zoom < 0.1f) zoom = 0.1f; // prevent zooming too close
	}
	else if (direction < 0) // scroll down
	{
		zoom += zoomStep;
		if (zoom > 10.0f) zoom = 10.0f;
	}
}
```

Mesh.h

```
#pragma once
#include <GL/glew.h>
#include <glm/glm.hpp>

#include <vector>
#include <cstdint>

class Mesh {
public:
    Mesh(
        const std::vector<glm::vec2>& vertices,
        const std::vector<glm::vec3>& colors,
        const std::vector<glm::vec2>& texCoords,
        const std::vector<std::uint32_t>& indices
    );
    ~Mesh();

    void Draw() const;
private:
            GLuint VAO, VBO[3], EBO;
            GLsizei indexCount;
};
```

Mesh.cpp

```
#include "./Mesh.h"

Mesh::Mesh(
	const std::vector<glm::vec2>& vertices,
	const std::vector<glm::vec3>& colors,
	const std::vector<glm::vec2>& texCoords,
	const std::vector<std::uint32_t>& indices
) {
   indexCount = static_cast<GLsizei>(indices.size());

   glGenVertexArrays(1, &VAO);
   glBindVertexArray(VAO);

   glGenBuffers(3, VBO);
   glGenBuffers(1, &EBO);

   // Vertex positions
   glBindBuffer(GL_ARRAY_BUFFER, VBO[0]);
   glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(glm::vec2), vertices.data(), GL_STATIC_DRAW);
   glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, nullptr);
   glEnableVertexAttribArray(0);

   // Colors
   glBindBuffer(GL_ARRAY_BUFFER, VBO[1]);
   glBufferData(GL_ARRAY_BUFFER, colors.size() * sizeof(glm::vec3), colors.data(), GL_STATIC_DRAW);
   glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, nullptr);
   glEnableVertexAttribArray(1);

   // Texture coordinates
   glBindBuffer(GL_ARRAY_BUFFER, VBO[2]);
   glBufferData(GL_ARRAY_BUFFER, texCoords.size() * sizeof(glm::vec2), texCoords.data(), GL_STATIC_DRAW);
   glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0, nullptr);
   glEnableVertexAttribArray(2);

   // Indices
   glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
   glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(uint32_t), indices.data(),
GL_STATIC_DRAW);

   glBindVertexArray(0);
}

Mesh::~Mesh() {
   glDeleteBuffers(3, VBO);
   glDeleteBuffers(1, &EBO);
   glDeleteVertexArrays(1, &VAO);
}

void Mesh::Draw() const {
   glBindVertexArray(VAO);
   glDrawElements(GL_TRIANGLES, indexCount, GL_UNSIGNED_INT, 0);
```

```
    glBindVertexArray(0);
}
```

Renderer.h

```
#pragma once
#include <GL/glew.h>
#include <GL/freeglut.h>

#include "Shader.h"
#include "../Scene.h"
#include "Mesh.h"
#include "../Entity.h"
#include "Camera.h"

#include <unordered_map>

//forward declarations
class App;

#define DEFAULT_CLEAR_COLOR_R 0.75f
#define DEFAULT_CLEAR_COLOR_G 1.0f
#define DEFAULT_CLEAR_COLOR_B 1.0f


class Renderer
{
public:
	Renderer(App* app);
	~Renderer();

	void Init();

	void Clear() const;

	std::shared_ptr<Mesh> AddMesh(const std::string& name, std::shared_ptr<Mesh> mesh);
	std::shared_ptr<Mesh> GetMesh(const std::string& name) const;

	std::shared_ptr<Shader> AddShader(const std::string& name, const std::string& vertPath, const
std::string& fragPath);
	std::shared_ptr<Shader> GetShader(const std::string& name) const;

	void Update() const;

	void RenderEntity(const Entity& entity) const;
	void Render() const;

	void Cleanup() const;

	void SetScene(Scene* newScene) { scene = newScene; }
	Scene* GetScene() const { return scene; }
	void SetCamera(Camera* newCamera) { camera = newCamera; }
	Camera* GetCamera() const { return camera; }

private:
	App* app;
```

```
        std::unordered_map<std::string, std::shared_ptr<Mesh>> meshCache;
        std::unordered_map<std::string, std::shared_ptr<Shader>> shaderCache;

        Shader* baseShader;

        Camera* camera = nullptr;

        Scene* scene = nullptr;

        //temporary static variables, move to another file later
        void CreateShaders(void);
};
```

Renderer.cpp

```
#include "./Renderer.h"

//temporary functions for rendering, move to another file later

#include "Util/loadShaders.h"

#include "../App.h"
#include "../InputManager.h"

void Renderer::CreateShaders(void)
{
	AddShader("slider", "shaders/slider.vert", "shaders/slider.frag");
	auto _baseShader = AddShader("base", "shaders/base.vert", "shaders/base.frag");
	baseShader = _baseShader.get();
}

Renderer::Renderer(App* app) : app(app)
{
	Init();
}

Renderer::~Renderer()
{
	meshCache.clear();
	shaderCache.clear();
}

void Renderer::Init()
{
	static Renderer* self = (Renderer*)this;

	glEnable(GL_BLEND);
	glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

	if (scene) {
		glClearColor(scene->backgroundColor.r, scene->backgroundColor.g, scene-
>backgroundColor.b, 1.0f);
	} else {
		glClearColor(DEFAULT_CLEAR_COLOR_R, DEFAULT_CLEAR_COLOR_G,
DEFAULT_CLEAR_COLOR_B, 1.0f);
	}

	glutDisplayFunc([]() {
		self->Update();
		self->Render();

		glutSwapBuffers();
		});

	glutIdleFunc([]() {
		glutPostRedisplay();
		});

	glutCloseFunc([]() {
		self->Cleanup();
		});

	glutKeyboardFunc([](unsigned char key, int x, int y) {
		if (key == 27) { // ESC key
			glutLeaveMainLoop();
		}
		InputManager::OnKeyPress(key);
		});


	glutMouseFunc([](int button, int state, int x, int y) {
		InputManager::OnMouseButton(button, state, x, y);
		});

	glutMotionFunc([](int x, int y) {
		InputManager::OnCursorPos(x, y);
		});

	glutPassiveMotionFunc([](int x, int y) {
		InputManager::OnCursorPos(x, y);
		});

	glutMouseWheelFunc([](int wheel, int direction, int x, int y) {
		InputManager::OnMouseWheel(direction);
		});

	CreateShaders();
}

void Renderer::Clear() const
{
	if (scene) {
		glClearColor(scene->backgroundColor.r, scene->backgroundColor.g, scene-
>backgroundColor.b, 1.0f);
	}
	else {
		glClearColor(DEFAULT_CLEAR_COLOR_R, DEFAULT_CLEAR_COLOR_G,
DEFAULT_CLEAR_COLOR_B, 1.0f);
	}
	glClear(GL_COLOR_BUFFER_BIT);
}

std::shared_ptr<Mesh> Renderer::AddMesh(const std::string& name, std::shared_ptr<Mesh> mesh)
{
	if (meshCache.find(name) == meshCache.end()) {
		meshCache[name] = mesh;
		return mesh;
	}
```

```
        return meshCache[name];
}

std::shared_ptr<Mesh> Renderer::GetMesh(const std::string& name) const
{
        auto it = meshCache.find(name);
        if (it != meshCache.end()) {
                return it->second;
        }
        return nullptr;
}

std::shared_ptr<Shader> Renderer::AddShader(const std::string& name, const std::string& vertPath, const
std::string& fragPath)
{
        if (shaderCache.find(name) == shaderCache.end()) {
                auto sh = std::make_shared<Shader>(vertPath.c_str(), fragPath.c_str());
                shaderCache[name] = sh;
                return sh;
        }
        return shaderCache[name];
}

std::shared_ptr<Shader> Renderer::GetShader(const std::string& name) const
{
        auto it = shaderCache.find(name);
        if (it != shaderCache.end())
                return it->second;
        return nullptr;
}

void Renderer::Update() const
{
        app->Update();
}

void Renderer::RenderEntity(const Entity& entity) const
{
        std::shared_ptr<Shader> useShader = entity.GetShader();
        if (!useShader)
                useShader = std::shared_ptr<Shader>(baseShader, [](Shader*) {}); // non-owning reference

        useShader->Use();

        if (entity.isGUI)
        {
                // GUI
                glm::mat4 view = glm::mat4(1.0f);
                glm::mat4 proj = glm::mat4(1.0f);

                useShader->setMat4("view", view);
                useShader->setMat4("projection", proj);
        }
        else
        {
                // Regular world-space entity
                useShader->setMat4("view", camera->GetViewMatrix());
                useShader->setMat4("projection", camera->GetProjectionMatrix());
        }

        entity.ApplyUniforms(*useShader);

        //draw the entity
        entity.Draw();
}

void Renderer::Render() const
{
        if (!camera || !scene) return;

        Clear();

        //do rendering here
        scene->Render();

        glFlush();
}

void Renderer::Cleanup() const
{
        //cleanup resources here
}
```

Shader.h

```
#pragma once
#include <GL/glew.h>
#include <GL/freeglut.h>
#include <glm/glm.hpp>


#include <string>

class Shader {
public:
    Shader(const char* vertexPath, const char* fragmentPath);
    ~Shader();

    void Use() const;
    GLuint GetID() const { return programID; }

    void setBool(const std::string& name, bool value) const;
    void setInt(const std::string& name, int value) const;
    void setFloat(const std::string& name, float value) const;
    void setVec3(const std::string& name, glm::vec3 value) const;
    void setMat4(const std::string& name, glm::mat4 value) const;

private:
    GLuint programID;

    static std::string LoadFile(const char* path);
    static void CheckCompileErrors(GLuint shader, const std::string& type);
};
```

Shader.cpp

```
#include "./Shader.h"

#include <glm/gtc/type_ptr.hpp>

#include <fstream>
#include <sstream>
#include <iostream>

Shader::Shader(const char* vertexPath, const char* fragmentPath) {
    std::string vertexCode = LoadFile(vertexPath);
    std::string fragmentCode = LoadFile(fragmentPath);
    const char* vCode = vertexCode.c_str();
    const char* fCode = fragmentCode.c_str();

    GLuint vertex = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertex, 1, &vCode, nullptr);
    glCompileShader(vertex);
    CheckCompileErrors(vertex, "VERTEX");

    GLuint fragment = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragment, 1, &fCode, nullptr);
    glCompileShader(fragment);
    CheckCompileErrors(fragment, "FRAGMENT");

    programID = glCreateProgram();
    glAttachShader(programID, vertex);
    glAttachShader(programID, fragment);
    glLinkProgram(programID);
    CheckCompileErrors(programID, "PROGRAM");

    glDeleteShader(vertex);
    glDeleteShader(fragment);
}

Shader::~Shader() {
    glDeleteProgram(programID);
}

void Shader::Use() const {
    glUseProgram(programID);
}

void Shader::setBool(const std::string& name, bool value) const
{
    glUniform1i(glGetUniformLocation(programID, name.c_str()), (int)value);
}
void Shader::setInt(const std::string& name, int value) const
{
    glUniform1i(glGetUniformLocation(programID, name.c_str()), value);
}
void Shader::setFloat(const std::string& name, float value) const
```

```
{
    glUniform1f(glGetUniformLocation(programID, name.c_str()), value);
}
void Shader::setVec3(const std::string& name, glm::vec3 value) const
{
    glUniform3fv(glGetUniformLocation(programID, name.c_str()), 1, glm::value_ptr(value));
}
void Shader::setMat4(const std::string& name, glm::mat4 value) const
{
    glUniformMatrix4fv(glGetUniformLocation(programID, name.c_str()), 1, GL_FALSE, glm::value_ptr(value));
}

std::string Shader::LoadFile(const char* path) {
    std::ifstream file(path);
    std::stringstream ss;
    ss << file.rdbuf();
    return ss.str();
}

void Shader::CheckCompileErrors(GLuint shader, const std::string& type) {
    GLint success;
    GLchar infoLog[1024];
    if (type != "PROGRAM") {
        glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
        if (!success) {
            glGetShaderInfoLog(shader, 1024, nullptr, infoLog);
            std::cerr << "ERROR::SHADER_COMPILATION_ERROR of type: " << type << "\n"
                << infoLog << "\n";
        }
    }
    else {
        glGetProgramiv(shader, GL_LINK_STATUS, &success);
        if (!success) {
            glGetProgramInfoLog(shader, 1024, nullptr, infoLog);
            std::cerr << "ERROR::PROGRAM_LINKING_ERROR\n"
                << infoLog << "\n";
        }
    }
}
```

Texture.h

```
#pragma once
#include <string>
#include <GL/glew.h>
#include <SOIL.h>

class Texture
{
public:
	Texture(const std::string& filePath);
	~Texture();

	void Bind(GLenum textureUnit = GL_TEXTURE0) const;
	GLuint GetID() const { return id; }
private:
	GLuint id = 0;
};
```

Texture.cpp

```
#include "./Texture.h"

#include <iostream>

Texture::Texture(const std::string& filePath)
{
    id = SOIL_load_OGL_texture(
        filePath.c_str(),
        SOIL_LOAD_AUTO,
        SOIL_CREATE_NEW_ID,
        SOIL_FLAG_INVERT_Y | SOIL_FLAG_MIPMAPS
    );

    if (!id)
    {
        std::cerr << "SOIL failed to load texture: " << filePath << "\n";
    }
    else
    {
        glBindTexture(GL_TEXTURE_2D, id);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        glBindTexture(GL_TEXTURE_2D, 0);

        std::cout << "Loaded texture: " << filePath << " (ID " << id << ")\n";
    }
}

Texture::~Texture()
{
    if (id)
        glDeleteTextures(1, &id);
}

void Texture::Bind(GLenum textureUnit) const
{
    glActiveTexture(textureUnit);
    glBindTexture(GL_TEXTURE_2D, id);
}
```

TextureManager.h

```
#pragma once
#include "Texture.h"

#include <unordered_map>
#include <memory>
#include <string>

//static class for managing textures
class TextureManager
{
public:
	static std::shared_ptr<Texture> Load(const std::string& path);
	static void Clear();
private:
	static std::unordered_map<std::string, std::shared_ptr<Texture>> textureCache;
};
```

TextureManager.cpp

```
#include "./TextureManager.h"

std::unordered_map<std::string, std::shared_ptr<Texture>> TextureManager::textureCache;

std::shared_ptr<Texture> TextureManager::Load(const std::string& path)
{
    auto it = textureCache.find(path);
    if (it != textureCache.end())
        return it->second; // reuse existing

    auto texture = std::make_shared<Texture>(path);
    textureCache[path] = texture;
    return texture;
}

void TextureManager::Clear()
{
    textureCache.clear();
}
```

Boid.h

```
#pragma once

#include "../Engine/Entity.h"

class Boid : public Entity
{
public:
	Boid(const glm::vec2& position = glm::vec2(0.0f), const glm::vec2& velocity = glm::vec2(0.0f));
	~Boid();

	glm::vec2 velocity;

	static void InitSharedResources(std::shared_ptr<Mesh> mesh, std::shared_ptr<Texture> texture);
private:
	static std::shared_ptr<Mesh> sharedMesh;
	static std::shared_ptr<Texture> sharedTexture;
};

float VectorToAngle(const glm::vec2& vec);
```

Boid.cpp

```
#include "./Boid.h"

std::shared_ptr<Texture> Boid::sharedTexture = nullptr;
std::shared_ptr<Mesh> Boid::sharedMesh = nullptr;

float VectorToAngle(const glm::vec2& vec)
{
	//return angle in degrees
	return atan2f(vec.y, vec.x) * 180.0f / 3.14159265f - 90.0f;
}

Boid::Boid(const glm::vec2& position, const glm::vec2& velocity)
	: Entity(sharedMesh, position, glm::vec2(0.25f), VectorToAngle(velocity), sharedTexture),
velocity(velocity)
{
}
Boid::~Boid()
{
}

void Boid::InitSharedResources(std::shared_ptr<Mesh> mesh, std::shared_ptr<Texture> texture)
{
	sharedMesh = mesh;
	sharedTexture = texture;
}
```

BoidScene.h

```
#pragma once

#include "../Engine/Scene.h"
#include "Boid.h"
#include "Nest.h"
#include "Cloud.h"
#include "Slider.h"

#include <unordered_map>

//forard declarations
class Mesh;
class Texture;

#define BOID_SCENE_W 20
#define BOID_SCENE_H 20

class BoidScene : public Scene
{
public:
	BoidScene(Renderer* renderer);
	~BoidScene();

	void InitBoids(uint16_t count);
	void InitNests(uint16_t count);
	void InitClouds(uint16_t count);

	static std::shared_ptr<Mesh> CreateBoidMesh();
	static std::shared_ptr<Mesh> CreateNestMesh();

	//override update function
	void SetUpdateFunction(const std::function<void(float)>& func) = delete;

	void AddBackground(std::shared_ptr<Mesh> mesh, std::shared_ptr<Texture> texture);
	void AddControlEntities(std::shared_ptr<Mesh> mesh);

	void ToggleMouseAttractAt(const glm::vec2& worldPos);
	void SetMouseAttractPos(const glm::vec2& worldPos);
	bool IsMouseAttractActive() const { return mouseAttractActive; }

	Entity* GetRandomBoid();
	Entity* GetRandomNest();
private:
	// Scene boundaries
	float leftBound = -BOID_SCENE_W / 2.0f;
	float rightBound = BOID_SCENE_W / 2.0f;
	float topBound = BOID_SCENE_H / 2.0f;
	float bottomBound = -BOID_SCENE_H / 2.0f;

	// Edge avoidance parameters
	float edgeMargin = 2.0f;	// Distance from edge before turning
```

```
        float turnFactor = 25.0f;     // How sharply boids turn near edges

        // Simulation parameters
        float neighborRadius = 1.0f;     // How far a boid can "see"
        float separationRadius = 0.5f;   // Minimum distance before repelling
        float minSpeed = 0.5f;           // Minimum speed
        float maxSpeed = 5.0f;           // Speed cap
        float maxForce = 5.0f;           // Steering force limit
        float alignmentWeight = 1.0f;
        float cohesionWeight = 0.4f;
        float separationWeight = 1.5f;

        float bias_increment = 0.0005f; // How much bias increases per update
        float max_bias = 0.05f;        // Maximum bias value

        float nestRadius = 0.5f;
        float nestAttractStrength = 3.0f;
        float nestRetargetInterval = 5.0f;  // seconds between random reassignments
        float timeSinceRetarget = 0.0f;
        std::unordered_map<const Boid*, int> boidNestMap; // which nest each boid is bound to

        bool mouseAttractActive = false;
        glm::vec2 mouseAttractPos = glm::vec2(0.0f);
        float mouseAttractRadius = 3.0f;
        float mouseAttractStrength = 30.0f;

        // Entities
        std::vector<Boid*> boidEntities;
        std::vector<Nest*> nestEntities;
        std::vector<Cloud*> cloudEntities;
        std::shared_ptr<Entity> backgroundEntity;
        Slider* alignmentControl, * cohesionControl, * separationControl;

        // Internal helpers
        std::vector<Boid*> GetNearbyBoids(const Boid* boid);
        glm::vec2 ComputeAlignment(const Boid* boid, const std::vector<Boid*>& neighbors);
        glm::vec2 ComputeCohesion(const Boid* boid, const std::vector<Boid*>& neighbors);
        glm::vec2 ComputeSeparation(const Boid* boid, const std::vector<Boid*>& neighbors);
        glm::vec2 ComputeNestAttraction(const Boid* boid);
        glm::vec2 ComputeMouseAttraction(const Boid* boid);
        void ApplyEdgeAvoidance(Boid* boid, float deltaTime);

        // Update logic
        void UpdateBoids(float deltaTime);
        void UpdateClouds(float deltaTime);
};
```

BoidScene.cpp

```
#include "./BoidScene.h"

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

#include <iostream>
#include <algorithm>

#include "../Engine/Renderer/Renderer.h"
#include "../Engine/Renderer/TextureManager.h"
#include "../Engine/InputManager.h"

BoidScene::BoidScene(Renderer* renderer) : Scene(renderer)
{
	backgroundColor = glm::vec3(0.375f, 0.7f, 1.0f);
	updateFunction = [this](float deltaTime)
		{
			UpdateBoids(deltaTime);
			UpdateClouds(deltaTime);
		};
}

BoidScene::~BoidScene()
{
}

void BoidScene::InitBoids(uint16_t count)
{
	//randomly add boids withing the scene bounds with random velocities and rotations
	for (uint16_t i = 0; i < count; ++i)
	{
		float x = static_cast<float>(rand() % (10*BOID_SCENE_W)) / 10.0f - BOID_SCENE_W / 2.0f;
		float y = static_cast<float>(rand() % (10*BOID_SCENE_H)) / 10.0f - BOID_SCENE_H / 2.0f;
		float speed = static_cast<float>((rand() % 25) + 25); // Speed between 25 and 50
		float angle = static_cast<float>(rand() % 360);
		glm::vec2 velocity = glm::vec2(cosf(glm::radians(angle)), sinf(glm::radians(angle))) * speed;
		auto boid = std::make_unique<Boid>(glm::vec2(x, y), velocity);
		boidEntities.push_back(boid.get());
		AddEntity(std::move(boid));

		// assign each boid to a random nest
		for (auto boid : boidEntities) {
			boidNestMap[boid] = rand() % nestEntities.size();
		}
	}
}

void BoidScene::InitNests(uint16_t count)
{
	//randomly add nests within the scene bounds
	for (uint16_t i = 0; i < count; ++i)
	{
		float x = static_cast<float>(rand() % (2 * (BOID_SCENE_W-1))) / 2.0f - (BOID_SCENE_W-1) /
2.0f;
		float y = static_cast<float>(rand() % (2 * (BOID_SCENE_H-1))) / 2.0f - (BOID_SCENE_H-1) /
2.0f;
		auto nest = std::make_unique<Nest>(glm::vec2(x, y));
		nestEntities.push_back(nest.get());
		AddEntity(std::move(nest));
	}
}

void BoidScene::InitClouds(uint16_t count)
{
	//randomly add clouds within the scene bounds
	for (uint16_t i = 0; i < count; ++i)
	{
		float x = static_cast<float>(rand() % (10 * BOID_SCENE_W)) / 5.0f - BOID_SCENE_W;
		float y = static_cast<float>(rand() % (10 * (BOID_SCENE_H+5))) / 10.0f - (BOID_SCENE_H+5) /
2.0f;
		float scale = static_cast<float>((rand() % 50) + 50) / 10.0f; // Scale between 5 and 10.0
		float opacity = static_cast<float>((rand() % 76) + 25) / 100.0f; // Opacity between 0.25 and
1.0
		float height = static_cast<float>((rand() % 50) + 50) / 50.0f; // Height between 1.0 and 2.0

		auto cloud = new Cloud(glm::vec2(x, y), glm::vec2(scale, scale), opacity, height, leftBound*2,
rightBound*2);
		cloudEntities.push_back(cloud);
	}

	// sort clouds by height so that higher clouds are rendered later
	std::sort(cloudEntities.begin(), cloudEntities.end(),
		[](const Cloud* a, const Cloud* b) {
			return a->getHeight() < b->getHeight();
		});
	for (auto& cloud : cloudEntities) {
		AddEntity(std::move(std::unique_ptr<Cloud>(cloud)));
	}
}

std::shared_ptr<Mesh> BoidScene::CreateBoidMesh()
{
	// Define a simple triangular mesh for the boid
	std::vector<glm::vec2> vertices = {
		{ 0.0f,  0.5f},
		{-0.5f, -0.5f},
		{ 0.0f, 0.0f},
		{ 0.5f, -0.5f },
	};
	std::vector<glm::vec3> colors = {
		{1.0f, 1.0f, 1.0f},
		{1.0f, 1.0f, 1.0f},
            {1.0f, 1.0f, 1.0f},
            { 1.0f, 1.0f, 1.0f }
    };
    std::vector<glm::vec2> texCoords = {
            { 0.5f,  1.0f},
            {0.0f, 0.0f},
            { 0.5f, 0.5f},
            { 1.0f, 0.0f },
    };
    std::vector<uint32_t> indices = { 0, 1, 2, 0, 3, 2 };
    return std::make_shared<Mesh>(vertices, colors, texCoords, indices);
}

std::shared_ptr<Mesh> BoidScene::CreateNestMesh()
{
    std::vector<glm::vec2> vertices = {
            {-0.5f,  0.0f},
            { 0.5f,  0.0f},
            { 0.3f, -0.25f},
            {-0.3f, -0.25f},
            {-0.4f,  0.0f},
            {-0.3f, 0.25f},
            {-0.2f, 0.25f},
            {-0.1f, 0.0f},
            { 0.2f, 0.0f},
            { 0.25f, 0.125f},
            { 0.35f, 0.125f},
            { 0.4f, 0.0f},

    };
    std::vector<glm::vec3> colors = {
            {1.0f, 1.0f, 1.0f},
            {1.0f, 1.0f, 1.0f},
            {1.0f, 1.0f, 1.0f},
            {1.0f, 1.0f, 1.0f},
            {1.0f, 1.0f, 1.0f},
            {1.0f, 1.0f, 1.0f},
            {1.0f, 1.0f, 1.0f},
            {1.0f, 1.0f, 1.0f},
            {1.0f, 1.0f, 1.0f},
            {1.0f, 1.0f, 1.0f},
            {1.0f, 1.0f, 1.0f},
            {1.0f, 1.0f, 1.0f},
    };
    std::vector<glm::vec2> texCoords = {
            {0.0f, 0.5f},
            {1.0f, 0.5f},
            {0.8f, 0.25f},
            {0.2f, 0.25f},
            {0.1f, 0.5f},
            {0.2f, 0.75f},
            {0.3f, 0.75f},
			{0.4f, 0.5f},
			{0.15f, 0.625f},
			{0.2f, 0.75f},
			{0.3f, 0.75f},
			{0.35f, 0.625f},
		};
		std::vector<uint32_t> indices = { 0, 1, 2, 0, 2, 3, 4, 5, 6, 4, 6, 7, 8, 9, 10, 8, 10, 11 };
		return std::make_shared<Mesh>(vertices, colors, texCoords, indices);
}

void BoidScene::AddBackground(std::shared_ptr<Mesh> mesh, std::shared_ptr<Texture> texture)
{
		backgroundEntity = std::make_shared<Entity>(mesh, glm::vec2(0.0f), glm::vec2(BOID_SCENE_W,
BOID_SCENE_H), 0.0f, texture);
		backgroundEntity->useTexture = true;
		AddEntity(std::unique_ptr<Entity>(backgroundEntity.get()));
}

void BoidScene::AddControlEntities(std::shared_ptr<Mesh> mesh)
{
		auto sliderShader = renderer->GetShader("slider");
		//get textures
		auto alignTex = TextureManager::Load("textures/alignment.png");
		auto cohTex = TextureManager::Load("textures/cohesion.png");
		auto sepTex = TextureManager::Load("textures/separation.png");

		float sliderWidth = 2.0f / 3.0f;
		// add 3 rectangles (scaled from a square mesh) at the bottom of the scene to control alignment,
cohesion, and separation weights
		alignmentControl = new Slider(mesh, glm::vec2(-sliderWidth, -0.9f), glm::vec2(sliderWidth, 0.2f), 0.0f,
alignTex, sliderShader, true, 0.0f, 2.0f);
		cohesionControl = new Slider(mesh, glm::vec2(0.0f, -0.9f), glm::vec2(sliderWidth, 0.2f), 0.0f, cohTex,
sliderShader, true, 0.0f, 2.0f);
		separationControl = new Slider(mesh, glm::vec2(sliderWidth, -0.9f), glm::vec2(sliderWidth, 0.2f), 0.0f,
sepTex, sliderShader, true, 0.0f, 2.0f);

		//controls for sliders
		alignmentControl->BindControl(alignmentWeight, '[', ']');
		cohesionControl->BindControl(cohesionWeight, ';', '\'');
		separationControl->BindControl(separationWeight, ',', '.');

		AddEntity(std::unique_ptr<Entity>(alignmentControl));
		AddEntity(std::unique_ptr<Entity>(cohesionControl));
		AddEntity(std::unique_ptr<Entity>(separationControl));
}

void BoidScene::ToggleMouseAttractAt(const glm::vec2& worldPos)
{
		if (!mouseAttractActive)
		{
			mouseAttractPos = worldPos;
			mouseAttractActive = true;
```

```
	}
	else
	{
		mouseAttractActive = false;
	}
}

void BoidScene::SetMouseAttractPos(const glm::vec2& worldPos)
{
	mouseAttractPos = worldPos;
}

glm::vec2 BoidScene::ComputeMouseAttraction(const Boid* boid)
{
	if (!mouseAttractActive) return glm::vec2(0.0f);

	glm::vec2 toMouse = mouseAttractPos - boid->position;
	float dist = glm::length(toMouse);
	if (dist <= 0.0001f) return glm::vec2(0.0f);

	if (dist > mouseAttractRadius) return glm::vec2(0.0f);

	// Strength falls off with distance
	float falloff = 1.0f - (dist / mouseAttractRadius); // 1 at center, 0 at radius
	glm::vec2 desired = glm::normalize(toMouse) * (mouseAttractStrength * falloff);
	return desired;
}

std::vector<Boid*> BoidScene::GetNearbyBoids(const Boid* boid)
{
	std::vector<Boid*> neighbors;
	for (const auto& entity : boidEntities)
	{
		if (entity != boid)
		{
			float dist = glm::length(entity->position - boid->position);
			if (dist < neighborRadius)
				neighbors.push_back(entity);
		}
	}
	return neighbors;
}

glm::vec2 BoidScene::ComputeAlignment(const Boid* boid, const std::vector<Boid*>& neighbors)
{
	if (neighbors.empty()) return glm::vec2(0.0f);
	glm::vec2 avgVel(0.0f);
	for (auto n : neighbors) avgVel += n->velocity;
	avgVel /= static_cast<float>(neighbors.size());
	avgVel = glm::normalize(avgVel) * maxSpeed;
	return avgVel - boid->velocity; // steering
}
```

```
glm::vec2 BoidScene::ComputeCohesion(const Boid* boid, const std::vector<Boid*>& neighbors)
{
	if (neighbors.empty()) return glm::vec2(0.0f);
	glm::vec2 center(0.0f);
	for (auto n : neighbors) center += n->position;
	center /= static_cast<float>(neighbors.size());
	glm::vec2 desired = center - boid->position;
	desired = glm::normalize(desired) * maxSpeed;
	return desired - boid->velocity;
}

glm::vec2 BoidScene::ComputeSeparation(const Boid* boid, const std::vector<Boid*>& neighbors)
{
	glm::vec2 steer(0.0f);
	for (auto n : neighbors)
	{
		float dist = glm::length(boid->position - n->position);
		if (dist < separationRadius && dist > 0)
		{
			glm::vec2 diff = glm::normalize(boid->position - n->position);
			steer += diff / dist; // stronger when closer
		}
	}
	if (glm::length(steer) > 0)
	{
		steer = glm::normalize(steer) * maxSpeed - boid->velocity;
	}
	return steer;
}

glm::vec2 BoidScene::ComputeNestAttraction(const Boid* boid)
{
	int nestIndex = boidNestMap[boid];
	Nest* nest = nestEntities[nestIndex];

	// Attraction towards nest center
	glm::vec2 toNest = nest->position - boid->position;
	float dist = glm::length(toNest);

	glm::vec2 attract(0.0f);
	if (dist > 0.01f) {
		attract = glm::normalize(toNest) * nestAttractStrength;

		//  if inside nest radius, orbit around it instead of heading straight in
		if (dist < nestRadius) {
			glm::vec2 tangent = glm::normalize(glm::vec2(-toNest.y, toNest.x)); // perpendicular
			attract += tangent * (nestAttractStrength * 0.5f); // swirl effect
		}
	}
	return attract;
}
```

```
void BoidScene::ApplyEdgeAvoidance(Boid* boid, float deltaTime)
{
	float distance;

	// Left edge
	distance = (boid->position.x - leftBound);
	if (distance < edgeMargin)
		boid->velocity.x += turnFactor * deltaTime * (1.0f - distance / edgeMargin);

	// Right edge
	distance = (rightBound - boid->position.x);
	if (distance < edgeMargin)
		boid->velocity.x -= turnFactor * deltaTime * (1.0f - distance / edgeMargin);

	// Bottom edge
	distance = (boid->position.y - bottomBound);
	if (distance < edgeMargin)
		boid->velocity.y += turnFactor * deltaTime * (1.0f - distance / edgeMargin);

	// Top edge
	distance = (topBound - boid->position.y);
	if (distance < edgeMargin)
		boid->velocity.y -= turnFactor * deltaTime * (1.0f - distance / edgeMargin);

}

void BoidScene::UpdateBoids(float deltaTime)
{
	timeSinceRetarget += deltaTime;
	if (timeSinceRetarget >= nestRetargetInterval) {
		timeSinceRetarget = 0.0f;

		// randomly reassign the nest of a random boid and its neighbors
		Entity* randomBoid = GetRandomBoid();
		if (randomBoid) {
			int newNestIndex = rand() % nestEntities.size();
			boidNestMap[static_cast<Boid*>(randomBoid)] = newNestIndex;
			auto neighbors = GetNearbyBoids(static_cast<Boid*>(randomBoid));
			for (auto neighbor : neighbors) {
				boidNestMap[neighbor] = newNestIndex;
			}
		}
	}


	for (auto& boid : boidEntities)
	{

		auto neighbors = GetNearbyBoids(boid);

		glm::vec2 align = ComputeAlignment(boid, neighbors) * alignmentWeight;
		glm::vec2 coh = ComputeCohesion(boid, neighbors) * cohesionWeight;
		glm::vec2 sep = ComputeSeparation(boid, neighbors) * separationWeight;
		glm::vec2 nestAttract = ComputeNestAttraction(boid);
		glm::vec2 mouseAttract = ComputeMouseAttraction(boid);

		glm::vec2 accel = align + coh + sep + nestAttract + mouseAttract;
		if (glm::length(accel) > maxForce)
			accel = glm::normalize(accel) * maxForce;

		boid->velocity += accel * deltaTime;

		ApplyEdgeAvoidance(boid, deltaTime);

		float speed = glm::length(boid->velocity);
		if (speed > 0.0f) // avoid division by zero
		{
			if (speed < minSpeed)
				boid->velocity = glm::normalize(boid->velocity) * minSpeed;
			else if (speed > maxSpeed)
				boid->velocity = glm::normalize(boid->velocity) * maxSpeed;
		}

		boid->position += boid->velocity * deltaTime;
		boid->rotation = VectorToAngle(boid->velocity);

	}
}

void BoidScene::UpdateClouds(float deltaTime)
{
	for (auto& cloud : cloudEntities)
	{
		cloud->Update(deltaTime);
	}
}

Entity* BoidScene::GetRandomBoid()
{
	if (boidEntities.empty()) return nullptr;
	int index = rand() % boidEntities.size();
	return boidEntities[index];
}

Entity* BoidScene::GetRandomNest()
{
	if (nestEntities.empty()) return nullptr;
	int index = rand() % nestEntities.size();
	return nestEntities[index];
}
```

Cloud.h

```
#pragma once
#include "../Engine/Entity.h"

#define CLOUD_SPEED_FACTOR 2.0f

class Cloud : public Entity
{
public:
	Cloud(const glm::vec2& position, const glm::vec2& scale, float opacity, float height, float leftBound, 
float rightBound);
	~Cloud();

	static void InitSharedResources(std::shared_ptr<Mesh> mesh, std::vector<std::shared_ptr<Texture>> 
textures);

	float getHeight() const { return height; }

	void Update(float deltaTime);
private:
	static std::shared_ptr<Mesh> sharedMesh;
	static std::vector<std::shared_ptr<Texture>> sharedTextures;

	float height;
	float leftBound, rightBound;
};
```

Cloud.cpp

```
#include "./Cloud.h"

std::shared_ptr<Mesh> Cloud::sharedMesh = nullptr;
std::vector<std::shared_ptr<Texture>> Cloud::sharedTextures = {};

template <typename T>
T RandomElement (const std::vector<T>& vec) {
	return vec[rand() % vec.size()];
}

Cloud::Cloud(const glm::vec2& position, const glm::vec2& scale, float opacity, float height, float leftBound,
float rightBound)
	: Entity(sharedMesh, position, scale, static_cast<float>(rand() % 360),
RandomElement(sharedTextures)), height(height), leftBound(leftBound), rightBound(rightBound)
{
	this->opacity = opacity;
}

Cloud::~Cloud()
{
}

void Cloud::InitSharedResources(std::shared_ptr<Mesh> mesh, std::vector<std::shared_ptr<Texture>>
textures)
{
	sharedMesh = mesh;
	sharedTextures = textures;
}

void Cloud::Update(float deltaTime)
{
	// Move cloud horizontally based on its height (higher clouds move faster)
	position.x += height * CLOUD_SPEED_FACTOR * deltaTime;
	// Wrap-around logic
	if (position.x - scale.x * 0.5f > rightBound) {
		position.x = leftBound - scale.x * 0.5f;
	}
}
```

Nest.h

```
#pragma once
#include "../Engine/Entity.h"

class Nest : public Entity
{
public:
	Nest(const glm::vec2& position);
	~Nest();

	static void InitSharedResources(std::shared_ptr<Mesh> mesh, std::shared_ptr<Texture> texture);
private:
	static std::shared_ptr<Mesh> sharedMesh;
	static std::shared_ptr<Texture> sharedTexture;
};
```

Nest.cpp

```
#include "./Nest.h"

std::shared_ptr<Texture> Nest::sharedTexture = nullptr;
std::shared_ptr<Mesh> Nest::sharedMesh = nullptr;

Nest::Nest(const glm::vec2& position)
	: Entity(sharedMesh, position, glm::vec2(0.25f), 0.0f, sharedTexture)
{
}

Nest::~Nest()
{
}

void Nest::InitSharedResources(std::shared_ptr<Mesh> mesh, std::shared_ptr<Texture> texture)
{
	sharedMesh = mesh;
	sharedTexture = texture;
}
```

Slider.h

```
#pragma once

#include "../Engine/Entity.h"

class Slider : public Entity
{
public:
	Slider(std::shared_ptr<Mesh> mesh, const glm::vec2& position = glm::vec2(0.0f), const glm::vec2&
scale = glm::vec2(1.0f), float rotation = 0.0f, std::shared_ptr<Texture> texture = nullptr,
std::shared_ptr<Shader> shader = nullptr,
		bool isGUI = false, float lower = 0.0f, float upper = 1.0f, float value = 0.0f);
	~Slider();

	float GetValue() const;
	void SetValue(float val);

	void ApplyUniforms(Shader& shader) const override;

	void BindControl(float& refVal, unsigned char keyLower, unsigned char keyUpper, float step = 0.1f);

private:
	float lowerBound;
	float upperBound;
	float value; // inbetween 0 and 1

};
```

Slider.cpp

```
#include "./Slider.h"

#include "../Engine/InputManager.h"

Slider::Slider(std::shared_ptr<Mesh> mesh, const glm::vec2& position, const glm::vec2& scale, float rotation,
std::shared_ptr<Texture> texture, std::shared_ptr<Shader> shader,
	bool isGUI, float lower, float upper, float value) : Entity(mesh, position, scale, rotation, texture,
shader, isGUI)
{
	if (upper < lower) {
		std::swap(lower, upper);
	}
	lowerBound = lower;
	upperBound = upper;
	this->value = (lowerBound == upperBound) ? 0 : (glm::clamp(value, lowerBound, upperBound) -
lowerBound) / (upperBound - lowerBound);
}

Slider::~Slider(){
}

float Slider::GetValue() const {
	return (lowerBound == upperBound) ? lowerBound : value * (upperBound - lowerBound) +
lowerBound;
}

void Slider::SetValue(float val) {
	value = (lowerBound == upperBound) ? 0 : (glm::clamp(val, lowerBound, upperBound) - lowerBound) /
(upperBound - lowerBound);
}

void Slider::ApplyUniforms(Shader& shader) const {
	Entity::ApplyUniforms(shader);

	shader.setFloat("sliderValue", value);

}

void Slider::BindControl(float& refVal, unsigned char keyLower, unsigned char keyUpper, float step) {
	InputManager::RegisterKeyAction(keyLower, [&refVal, this, step]() {
		refVal -= step;
		refVal = glm::clamp(refVal, lowerBound, upperBound);
		SetValue(refVal);
		});
	InputManager::RegisterKeyAction(keyUpper, [&refVal, this, step]() {
		refVal += step;
		refVal = glm::clamp(refVal, lowerBound, upperBound);
		SetValue(refVal);
		});

	//set value to the reference so there is no potential discrepancy initially
```

```
        SetValue(refVal);
        //
}
```

base.vert

```
#version 330 core

layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;
layout (location = 2) in vec2 aTexCoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

out vec3 vColor;
out vec2 vTexCoord;

void main()
{
    vColor = aColor;
    vTexCoord = aTexCoord;

    gl_Position = projection * view * model * vec4(aPos, 0.0, 1.0);
}
```

base.frag

```
#version 330 core

in vec3 vColor;
in vec2 vTexCoord;

uniform sampler2D tex;
uniform bool useTexture;
uniform float opacity;

out vec4 FragColor;

void main()
{
    vec4 texColor = vec4(1.0);
    if (useTexture)
        texColor = texture(tex, vTexCoord);

    FragColor = texColor * vec4(vColor, opacity);
}
```

slider.vert

```
#version 330 core

layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;
layout (location = 2) in vec2 aTexCoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

out vec2 vTexCoord;

void main()
{
   vTexCoord = aTexCoord;

   gl_Position = projection * view * model * vec4(aPos, 0.0, 1.0);
}
```

slider.frag

```
#version 330 core
out vec4 FragColor;

in vec2 vTexCoord;

uniform sampler2D tex;

uniform float sliderValue = 0.5;

uniform vec3 FillColor = vec3(0.0, 1.0, 0.0);
uniform vec3 BackgroundColor = vec3(1.0, 0.0, 0.0);
uniform vec3 BorderColor = vec3(0.0, 0.0, 1.0);

void main()
{
    if(vTexCoord.x < 0.05 || vTexCoord.x > 0.95 || vTexCoord.y < 0.05 || vTexCoord.y > 0.95)
        FragColor = vec4(BorderColor, 0.1);
    else if (vTexCoord.x <= sliderValue * 0.90 + 0.05)
        FragColor = vec4(FillColor, 0.5);
    else
        FragColor = vec4(BackgroundColor, 0.5);

    vec4 TexColor = texture(tex, vTexCoord);
    FragColor *= TexColor;
}
```