Tutorial – Fixed Timestep

Introduction and Objective:

In this tutorial, we will begin creating a custom 2D physics engine. Over the next several sessions we will build on these tutorials, adding features as we discuss new concepts during class.

This first tutorial will set up the application and a physics scene using a fixed timestep.

At the end of this tutorial you will not have a working program. But you will have a codebase you can use when working through the next tutorial, should you choose to use it. Alternatively, you may choose to ignore this tutorial and start from scratch in the tutorial for the session on *Linear Force and Momentum*.

Required Resources:

We begin building off the AIE Bootstrap framework 2D template.

Ensure you have an updated version of bootstrap, and have installed the Visual Studio project templates.

Process:

Create a new 2D Bootstrap project and ensure it will compile and run.

Our objective in this tutorial is setting up a *PhysicsScene* class. This class will control the simulation and drawing of all physical objects in our game.

Create a new class called *PhysicsScene* that has the following interface.

```
class PhysicsScene
{
public:
	PhysicsScene();
	~PhysicsScene();

	void addActor(PhysicsObject* actor);
	void removeActor(PhysicsObject* actor);
	void update(float dt);
	void updateGizmos();

	void setGravity(const glm::vec2 gravity) { m_gravity = gravity; }
	glm::vec2 getGravity() const { return m_gravity; }

	void setTimeStep(const float timeStep) { m_timeStep = timeStep; }
	float getTimeStep() const { return m_timeStep; }

protected:
	glm::vec2 m_gravity;
	float m_timeStep;
	std::vector<PhysicsObject*> m_actors;
};
```

You will need to include the *glm vector2* class header, as well as the *std vector* class header, and forward declare the *PhysicsObject* class.

This class should be relatively straightforward at this point.

We have a *vector* collection (*m*_actors) that stores all the physical objects in the scene, along with functions to add and remove objects to and from this list.

You will notice that we have two *update* functions. One called *update()*, and one called *updateGizmos()*.

The *update()* function will update the physical simulation. It will call the *update* function of each actor, ensuring that the actors position is updated according to its internal state. It will also be where we handle collision detection and response.

The *updateGizmos()* function will be called by the application once per frame. Essentially, this function will be responsible drawing the physical objects. Because we are batching our draw calls we want to add each *gizmo* to the render using an *add* function call during the update loop. Then, when drawing, the renderer can draw all objects at once.

If you explore the *Gizmos* class in the bootstrap framework, you will see functions for adding a variety of objects, like circles and lines, to the renderer. *Gizmos* also has a *draw2D* function, which draws all gizmos added during the last update.

We won't be implementing any physical objects for the application to simulate or draw during this tutorial, but it is important to understand how the application is structured so that you can proceed with future tutorials.

Before continuing ensure you have implemented the following functions of the *PhysicsScene* class:

- constructor: initialize the timestep (0.01f) and gravity (0,0).
- addActor(): adds a *PhysicsObject* pointer to the end of the *m_actors* vector.
- removeActor(): removes a *PhysicsObject* pointer from the *m_actors* vector.

We'll discuss how to implement the *PhysicsScene::update()* and *PhysicsScene::updateGizmos()* functions later in this tutorial.

Before we implement the two *update* functions, we'll create the definition of the *PhysicsObject* class.

```
class PhysicsObject
{
protected:
	PhysicsObject() {}

public:
	virtual void fixedUpdate(glm::vec2 gravity, float timeStep) = 0;
	virtual void debug() = 0;
	virtual void makeGizmo() = 0;
	virtual void resetPosition() {};
};
```

PhysicsObject is a pure abstract base class.

We won't be deriving anything from it during this tutorial. We simply define the class so that our *PhysicScene* class implementation is complete.

In future sessions you will define shapes, like AABBs and circles, that derive from this class.

Return to the .cpp file for the *PhsyicsScene* class and add the following function implementations:

```
PhysicsScene::PhysicsScene() : m_timeStep(0.01f), m_gravity(glm::vec2(0, 0))
{
}

void PhysicsScene::update(float dt)
{
	// update physics at a fixed time step

	static float accumulatedTime = 0.0f;
	accumulatedTime += dt;

	while (accumulatedTime >= m_timeStep)
	{
		for (auto pActor : m_actors)
		{
			pActor->fixedUpdate(m_gravity, m_timeStep);
		}

		accumulatedTime -= m_timeStep;
	}
}

void PhysicsScene::updateGizmos()
{
	for (auto pActor : m_actors) {
		pActor->makeGizmo();
	}
}
```

In the *PhysicsScene::update()* function, we've implemented a fixed timestep using the pseudocode outlined in the lecture slides.

Note that in future tutorials you will need to keep track of the previous and current position of each physics object so that you can interpolate between the two states when drawing. While this is not essential, failing to do this will mean that your simulation will stutter slightly – a problem known as *temporal aliasing*.

The final task is to integrate our *PhysicsScene* class into our application class.

Add a new *PhsicsScene** variable to your application class called *m_physicsScene*, and update your application's functions as follows:

```
bool Physics00_FixedTimestepApp::startup() {
	// increase the 2d line count to maximize the number of objects we can draw
	aie::Gizmos::create(255U, 255U, 65535U, 65535U);

	m_2dRenderer = new aie::Renderer2D();
	m_font = new aie::Font("./font/consolas.ttf", 32);

	m_physicsScene = new PhysicsScene();
	m_physicsScene->setTimeStep(0.01f);
	return true;
}

void Physics00_FixedTimestepApp::update(float deltaTime) {
	// input example
	aie::Input* input = aie::Input::getInstance();

	aie::Gizmos::clear();

	m_physicsScene->update(deltaTime);
	m_physicsScene->updateGizmos();

	// exit the application
	if (input->isKeyDown(aie::INPUT_KEY_ESCAPE))
		quit();
}

void Physics00_FixedTimestepApp::draw() {
	// wipe the screen to the background colour
	clearScreen();

	// begin drawing sprites
	m_2dRenderer->begin();

	// draw your stuff here!
	static float aspectRatio = 16 / 9.f;
	aie::Gizmos::draw2D(glm::ortho<float>(-100, 100,
		-100 / aspectRatio, 100 / aspectRatio, -1.0f, 1.0f));

	// output some text, uses the last used colour
	m_2dRenderer->drawText(m_font, "Press ESC to quit", 0, 0);
	// done drawing sprites
	m_2dRenderer->end();
}
```

If you can't find the header file that includes the *glm::ortho* function, it's in <glm\ext.hpp>.

After creating the *PhysicsScene* object, we set the timestep to 0.01. Decreasing the value of this timestep will increase the accuracy of our physic simulation at the expense of increased processing time. Setting this value too high will make out simulation stutter, in addition to reducing the stability of the simulation.

The *update()* function will call each *update* function of the *PhysicsScene* class in turn. And the *draw()* function will simply call *Gizmos::draw2D()* to draw any gizmos added to the renderer during the last update (gizmos are cleared every frame via the call to *Gizmos::clear()* in the *update()* function).

Compile your work. You should be able to compile without errors (ensure you added the implementation of the *addActor* and *removeActor* functions in the *PhysicsScene* class yourself).

Even if you execute your program, at this state all you will see is a blank screen. However, our application is now set up to begin the next tutorial.

As you complete future tutorials, you may wish to modify them to allow the renderer to interpolate between the current and past positions of your physics objects so as to adjust for any temporal aliasing.