2D Physics Engine

Computer Science Non-Examined Assessment

Scott Richmond-Wood

Contents

Analysis

Background

Simulation is a large area of computer science, from games design to more specialist simulations of aerodynamics, particle collisions, etc. We use computers to simulate all of these. And at the core of all simulations is a physics engine, which is used to model the physics of the given environment. As such, the emulation of physics through programming is a very important idea which allows us to model many theories or to create different environments.
Therefore, I've decided to investigate how physics is simulated in a 2 dimensional environment and create a 2D physics engine. This will require abstraction from physics in a 3 dimensional environment as I am working with only an x-axis and a y-axis. And thus will allow me to focus more on creating a complete physical environment.
Furthermore, the functionality of the engine needs to be demonstrated on an interface. This means extra components are required: a rendering pipeline and an interface on which to run it.
Since I'm not working in 3 dimensions there are some differences. A 3d engine is more realistic as our world is 3d but in some cases 3d is unneeded for example testing how uniform prisms act can be done in 2d or 3d but 2d is less computationally expensive so is useful in such cases. Also working in 3d can be very complex so 2d allows me to cover more laws of physics within the time I'm given. It also gives me a chance to remove certain aspects from the laws in 3d and reduce them to work in a 2d environment and help me gain a greater understanding of these laws.
Whilst my project is primarily investigatory, I am creating this engine for an indie gaming company called NanoShark.

My Client

NanoShark is a recently formed indie gaming company run by a group of students. They are currently working on a few different games using the Unity engine. Whilst Unity is a very good game engine for learning and taking care of environment creation, the company wants its own custom engine to work with. As such they want a physics engine to work with the rest of the components, very few of the members have ever coded an application of this scale from scratch so have approached me to help them create the physics engine.

Systems Research

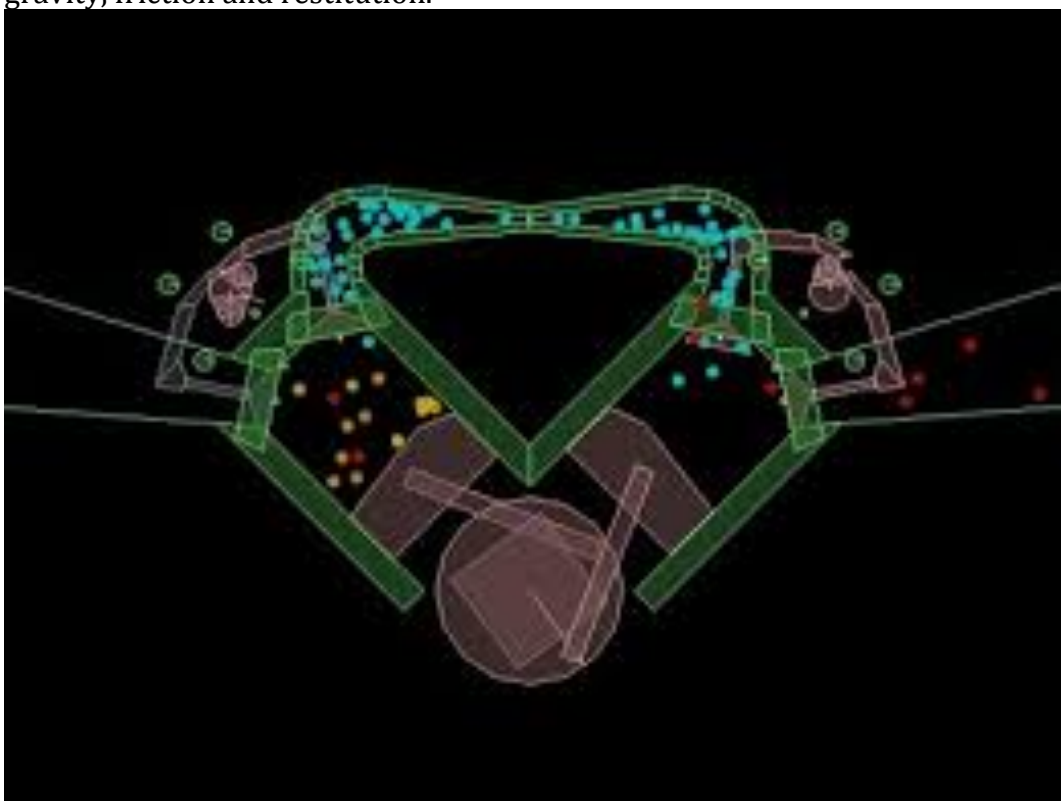
Existing Systems

Most engines today contain their own physics, they all vary based on the purpose of the engine. The variation between different engines is vast so to try and narrow down the possibilities I researched engines that worked predominantly in 2 dimensions as opposed to 3.

- **Box2D**: Box2D is a physics engine that was specifically designed for developing games. Its physics are based around rigid body simulation, this means it assumes the objects do not deform under the application of a force and instead retain their shape. This means it can focus on how the system of objects (bodies) moves when the force is applied.

It deals with collisions by an incremental broad phase (utilising an algorithm called sweep and prune) which checks which objects may be colliding followed by a narrow phase that checks which objects are colliding and resolves the collision.

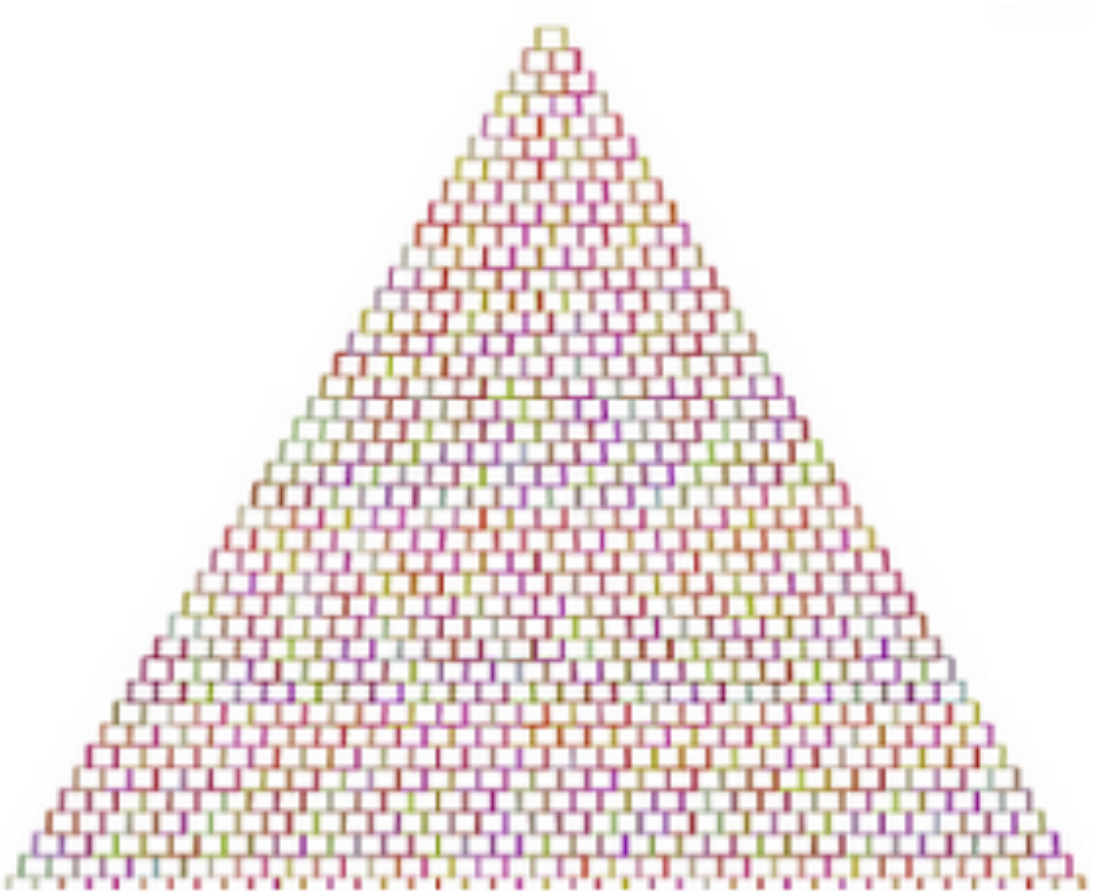
Box2D is able to simulate objects which are comprised of convex polygons (interior angles < 180), circles and edge shapes, all of which can rotate. They are joined together by joints and acted upon by forces. It also applies gravity, friction and restitution.



Picture of a combustion engine, constructed using the Box2D engine.

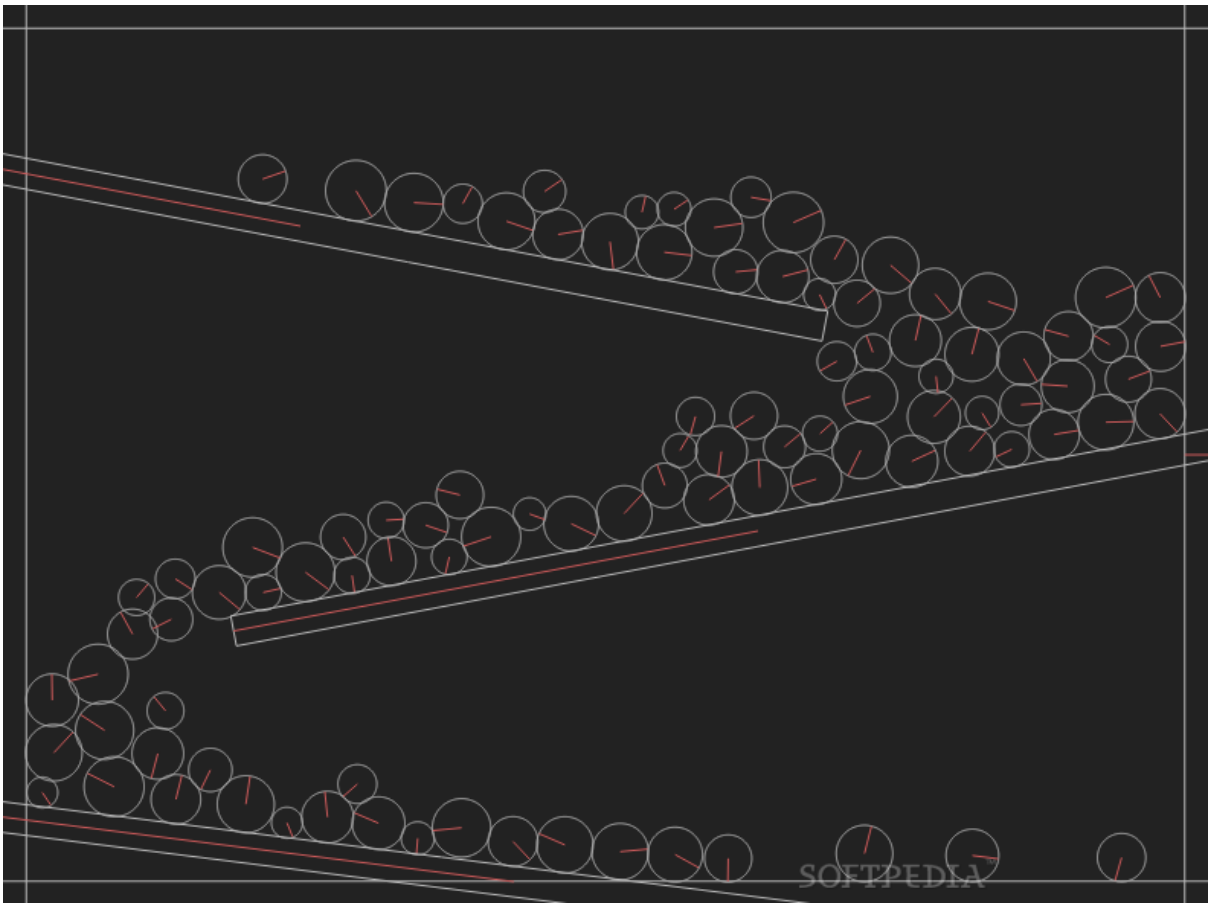
- **Nape:** Nape is a more general engine and can be used for a variety of purposes. Once again, the physics are based around rigid body simulation. It comes with buoyancy physics meaning it can deal with rigid bodies which are submerged in liquid.
  The objects are comprised of convex polygons and circles, which can rotate.

  It continually checks for collisions, which avoids the possibility of any intersections bypassing the detection and resolution algorithms. It also allows constraints to be placed on the objects position, velocity, etc. The application of gravity, friction and restitution are also covered.



A stable pyramid showcasing the contact forces of the Nape Physics Engine

- **Matter.js:** Matter.js is a physics engine that was created for use with JavaScript making it easy to integrate with web development as well as a range of other languages (e.g. in unity JS is used in conjunction with C#). It works with rigid bodies and compositions of rigid bodies. Allows for physical properties such as mass and density, deals with restitution and has friction and collision resolution. On top of which it allows the user to affect the flow of time (i.e. slow-motion and sped-up). The fact that it's written in JS means it has cross-browser compatibility and is mobile compatible.



An image of various sized circles rolling down a set of ramps in Matter.js

Interview

I also interviewed one of the company's co-founders Mr Kingsbury in order to grasp what NanoShark wanted of their engine. Interview details are given below.

- What kind of games are you looking to work on?

  Whilst we have many games planned, our primary concern as of late is creating 2-dimensional, plane-locked games. Due to our small budgets we are unable to garner the graphical depth and fidelity we feel is acceptable within today's market within a viable timeframe, therefore most of our planned projects are focused on the core mechanics and back-end of our projects. The engine we use must reflect this.

- Why do you want a custom physics engine as opposed to one of the many that are already out there?

  We at NanoShark believe a more bespoke engine will give us a far more optimised and simplistic approach to our in-engine physics. There are also significant legal and licencing issues attached to the release of products using commercial products we hope to circumvent most of this by using a more niche product. The use of a more bespoke engine will allow us to have far more control over specific features, in most commercial products there is a need to generalise their use to reach a majority audience, we believe if we have some say in the way it [the engine] is written we may be able to cut out unnecessary confusion.

- What should a physics engine ideally cover for you?

  Ideally it will be able to handle basic collisions between multiple defined hit boxes. It must also have some form of dynamic friction on entities within the engine. Ideally this would scale with force applied to the entity though this is not a necessity. It must also cover the basics of mass in relation to gravity and ways to scale this. If possible it should also be able to demonstrate simple harmonic motion in some capacity.

Summary of Research

Name	Gravity	Collision Resolution	Friction	Restitution	Rotation	Time Manipulation	Platforms	Body Type
Box2D	Yes	Yes	Yes	Yes	Yes	No	PC, Mac, Web, PS3, PS4, WiiU, Xbox 360, Xbox One	Rigid
Nape	Yes	Yes	Yes	Yes	Yes	No	PC, Mac, Web	Rigid
Matter.JS	Yes	Yes	Yes	Yes	Yes	Yes	PC, Mac, Web, Mobile	Rigid
Interview	Yes	Yes	Yes	Not Specified	Not Specified	Not Specified	Not Specified	Rigid

The engines may have their different purposes but some similarities stand out:

- Gravity
- Friction
- Restitution
- Collision Resolution
- Rotation
- Rigid Bodies

These points are covered by every engine that has been researched and are integral to the production of a physics engine. As such these would make a good basis on which to base my project, despite not all of them having been specified by my client. The engine's capabilities also need to be demonstrated on an interface, so this will require a simple rendering pipeline in order to render the scene. There also need to be some shapes with which the engine can deal, using lines (very thin rectangles) and circles I can create any convex polygon and thus demonstrate the physics on a variety of objects.

Objectives

- Should be able to detect collision between objects, which is done by representing objects as points and checking if the points overlap.
  This will be done by:
    a. Representing a shape in one of two ways
        i) A circle will be represented as a centre point and a radius
        ii) Other polygons will have their vertices co-ordinates stored in a list
        iii) As such circles and polygons will be different classes
    b. Checking if the points overlap using conditional statements
  This will be measured by:
    a. Conducting tests where shapes collide in different ways and checking if the collision detection steps occur

- Every object will have some base properties, which could differ slightly from object to object. So I'll need a way to organise the data.
  This will be done by:
    a. Having a variety of classes for different types of object.
    b. Separating classes into files based on similarity
    c. Separating files into namespaces based on similarity

- Should be able to resolve collisions, which I plan to do via impulse resolution.
  This will be done by:
    a. Detecting a collision as described above
    b. Taking data about the colliding objects
    c. Using the velocities and masses to solve a momentum equation
    d. Applying the new velocities to the objects
  This will be measured by:
    a. Conducting tests where shapes collide in different ways and checking if they react in the intended way (i.e. hitting a wall causes an object to bounce off or stop depending on its speed)

- As a part of collision resolution I will have to apply friction to objects when they're rubbing against one another. Which is done via finding a coefficient of friction and a tangent vector and using some formulae from classical mechanics.
  This will be done by:
    a. Calculating a coefficient of friction using a formula based on the roughness of each object
    b. Finding a tangent vector to the collision normal
    c. Applying friction based on $F \leq \mu R$ (Friction is less than or equal to the product of the coefficient of friction and the normal reaction.
  This will be measured by:
    a. Conducting tests where shapes rub against one another in different scenarios (down a wall or down a ramp) and checking

if the object slips or sticks as it should based on my calculations.

- I will also need to know which types of objects are colliding so I know which collision method to call
  This will be done by:
  a. Having each object have a shape type property which will describe the shape
  b. Using conditional statements using collision detection to check this property and using the correct method
  This will be measured by:
  a. Conducting tests where different combinations of shapes collide and checking whether the point at which the engine takes action is correct.

- The objects need to be able to move around with a linear velocity and acceleration
  This will be done by:
  a. Getting the forces applied
  b. Calculating acceleration based on acceleration = force / mass
  c. Calculating velocity based on velocity += acceleration * time
  d. Calculating displacement based on displacement += velocity * time
  This will be measured by:
  a. Conducting tests of acceleration such as an object in free fall and noting whether the object accelerates and moves

- The scene needs to be rendered using a basic pipeline
  This will be done by:
  a. Queuing objects that need to be rendered
  b. Using rendering libraries to render each object
  This will be measured by:
  a. Launching the program and seeing if the objects are displayed
  b. Drawing new objects and seeing if they are rendered

- A variety of scenes are needed in order to test each aspect, so in order to avoid creating many different set scenes, I will incorporate the ability to draw polygons and circles.
  This will be done by:
  Circles:
  a. Having a circle drawing mode, toggled by a key
  b. Using the mouse position to determine the desired radius
  c. Creating a new circle instance
  d. Storing a radius and a centre position
  Polygons:
  a. Having a polygon drawing mode, toggled by a key
  b. Gathering points based on a mouse click and storing them in a list

c. Creating a new polygon instance with this list of points

This will be measured by:

a. Testing if the respective drawing modes toggle on input
b. Testing if drawing a circle creates a circle and changes size based on dragging the mouse
c. Testing if I can place points for a polygon
d. Testing that when I turn off the drawing mode the object is created

Object Model

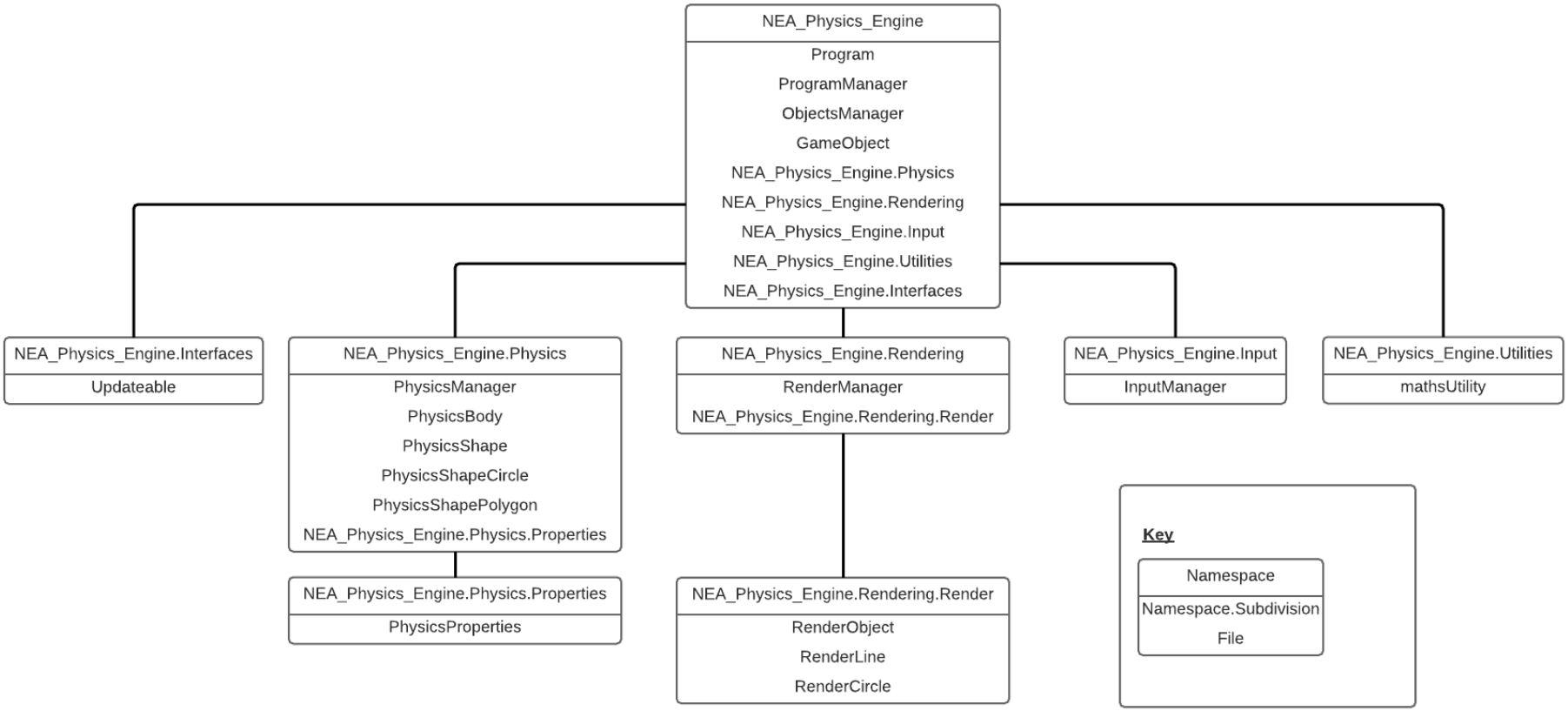
I will be using C# in order to create the engine, thus I can store multiple classes under a namespace, and this is the highest level in which I can visualise my engine. By storing classes under namespaces I can use large quantities of content by referencing a namespace. The following diagram shows how the various namespaces are related.

The diagram shows the namespace NEA_Physics_Engine which encompasses the entirety of the program and makes sure it is all accessible. It directly contains the Program which initiates the engine, the ProgramManager which uses the other managers to run the program and also takes care of the drawing functionality, the ObjectsManager which manages the list of objects which exist within the program (when a new object is created it adds it to the list etc.) and the GameObject which has an instance for each object and initialises it and updates its properties. It also contains a number of namespaces.

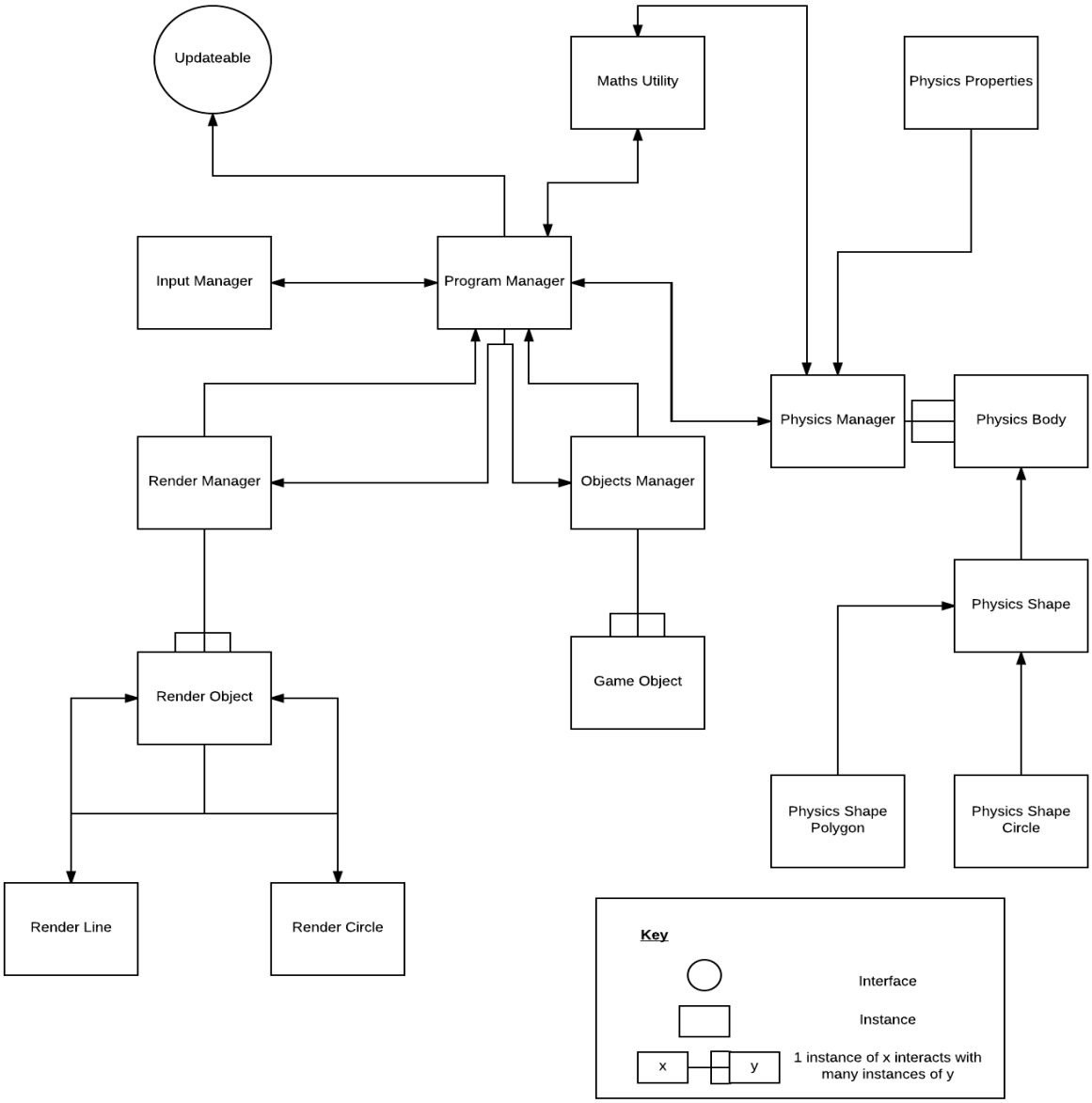
The next level down contains a number of namespaces. The NEA_Physics_Engine.Interfaces namespace contains Updateable which is simply an interface for the program to use. The NEA_Physics_Engine.Input namespace contains the InputManager which detects inputs from the user and uses them as Booleans or Vectors dependent on the input type (for example if the left mouse button is clicked then it sets a Boolean condition of LeftMouseDown to true). The NEA_Physics_Engine.Utilities namespace contains the mathsUtility which contains a number of mathematical formulae which are used throughout the program.

The other two namespaces on this level break down even further. NEA_Physics_Engine.Physics contains the PhysicsManager which applies the physics to each object and manages the list of objects which have physics, the PhysicsBody which deals with the displacement, velocity and acceleration of the object, the PhysicsShape which sets up an objects collider, PhysicShapeCircle which sets up a circle's physical properties and PhysicsShapePolygon which sets up a polygon's physical properties. It also contains the namespace NEA_Physics_Engine.Physics.Properties which contains PhysicsProperties which contains all the constraints on values such as maximum velocity and material.

NEA_Physics_Engine.Rendering contains the RenderManager which queues and renders the objects and the namespace NEA_Physics_Engine.Rendering.Render. This namespace contains RenderObject which sets the texture of the object, RenderCircle which sets the rendering properties of a circle and RenderLine which sets the rendering properties of the edge of a polygon.


NEA_Physics_Engine
Program
ProgramManager
ObjectsManager
GameObject
NEA_Physics_Engine.Physics
NEA_Physics_Engine.Rendering
NEA_Physics_Engine.Input
NEA_Physics_Engine.Utilities
NEA_Physics_Engine.Interfaces
NEA_Physics_Engine.Interfaces
Updateable
NEA_Physics_Engine.Physics
PhysicsManager
PhysicsBody
PhysicsShape
PhysicsShapeCircle
PhysicsShapePolygon
NEA_Physics_Engine.Physics.Properties
NEA_Physics_Engine.Physics.Properties
PhysicsProperties
NEA_Physics_Engine.Rendering
RenderManager
NEA_Physics_Engine.Rendering.Render
NEA_Physics_Engine.Rendering.Render
RenderObject
RenderLine
RenderCircle
NEA_Physics_Engine.Input
InputManager
NEA_Physics_Engine.Utilities
mathsUtility
Key
Namespace
Namespace.Subdivision
File

There are many files within each namespace, which will interact with one another in order to create the engine itself. The way in which these are related is more complex than the simple namespace relation, as each file can have multiple instances and thus can occur multiple times for a single instance of another file. The way in which these instances are related is given below.
The Program Manager displays the program on the Updateable interface and calls the Maths Utility to perform vector calculations, it also calls the input manager to detect any relevant inputs. On top of this it calls the Render Manager to render new and updated objects, the Objects Manager to update the list of objects when a new object is created or removed and the Physics Manager to deal with the physics of each object.
The Render Manager has many instances of Render Object as many objects need to be rendered and each Render Object is either a line or a circle so Render Object calls Render Line or Render Circle respectively.
The Objects Manager stores a list of objects each of which is an instance of Game Object.
The Physics Manager needs to know the constraints on the physics it's applying so calls Physics Properties. It also calls the Maths Utility for vector and volume calculations. Every object the Physics Manager affects is an instance of Physics Body which has a collider from Physics Shape and the collider is determined by its shape so Physics Shape calls the corresponding file.


Updateable
Maths Utility
Physics Properties
Input Manager
Program Manager
Physics Manager
Physics Body
Render Manager
Objects Manager
Physics Shape
Game Object
Render Object
Physics Shape Polygon
Physics Shape Circle
Render Line
Render Circle
Key
Interface
Instance
x
y
1 instance of x interacts with many instances of y

Design

Purpose Of The System

The purpose of this system is to simulate rigid body dynamics in a realistic way in order for it to be used in a custom game engine. As such it will simulate classical mechanics, this includes acceleration, forces, impulses, collisions, damping (i.e. air resistance) and friction.
The system should be easy to use and understand, as it will likely be edited in the future.

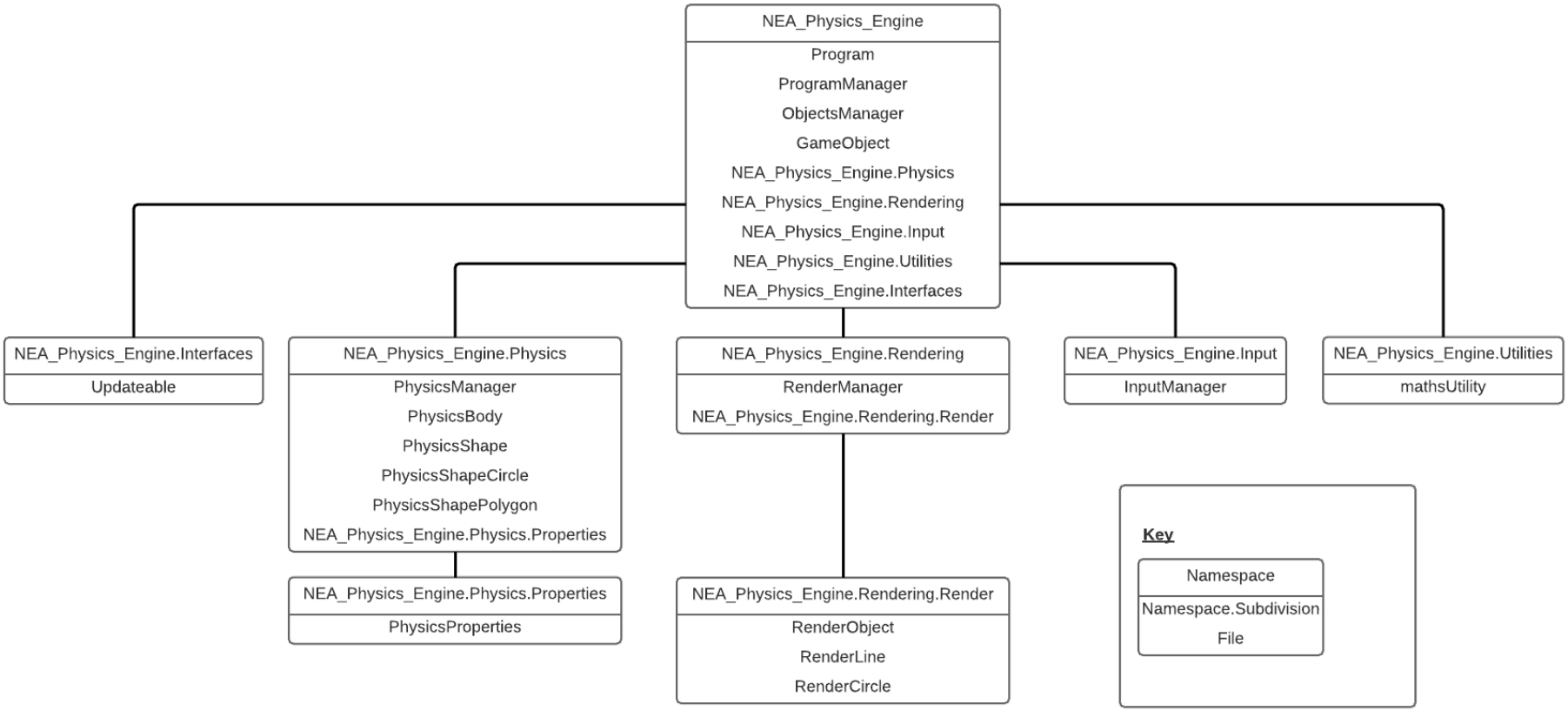
Runtime Environment

- Windows 7 (or newer): The engine will be using framework that is only available on Windows systems, which is acceptable as the company uses Windows systems to create their games and is primarily looking to distribute games on Windows systems.
- XNA Framework 4.0: I'll be using the libraries included in XNA Game Studio 4.0, which means any solution will require the XNA framework to run.

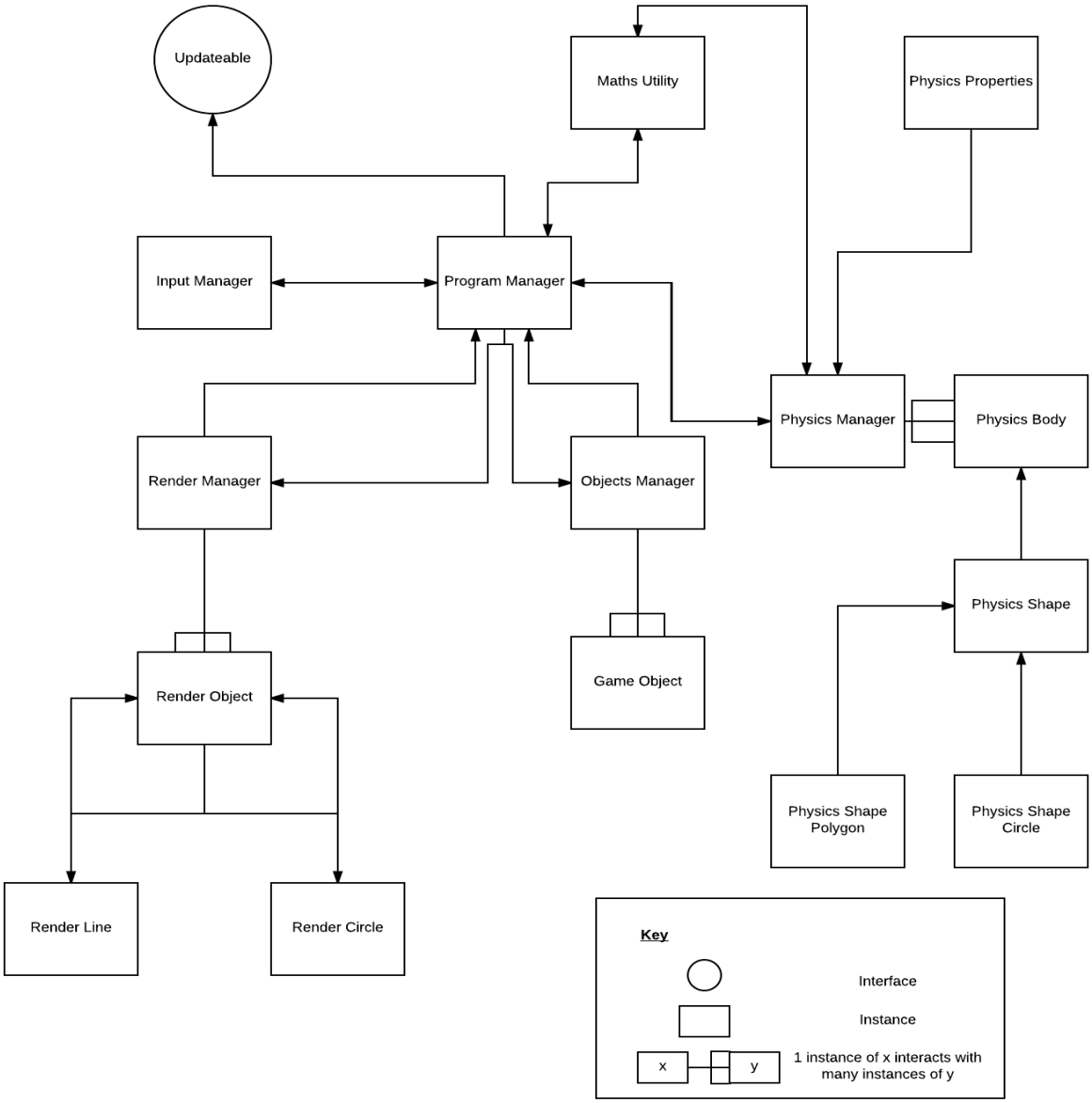
Extension Of Object Model

I will be using C# in order to create the engine, thus I can store multiple classes under a namespace, and this is the highest level in which I can visualise my engine. By storing classes under namespaces I can use large quantities of content by referencing a namespace. The following diagram shows how the various namespaces are related.
The diagram shows the namespace NEA_Physics_Engine which encompasses the entirety of the program and makes sure it is all accessible. It directly contains the Program which initiates the engine, the ProgramManager which uses the other managers to run the program and also takes care of the drawing functionality, the ObjectsManager which manages the list of objects which exist within the program (when a new object is created it adds it to the list etc.) and the GameObject which has an instance for each object and initialises it and updates its properties. It also contains a number of namespaces.
The next level down contains a number of namespaces. The NEA_Physics_Engine.Interfaces namespace contains Updateable which is simply an interface for the program to use. The NEA_Physics_Engine.Input namespace contains the InputManager which detects inputs from the user and uses them as Booleans or Vectors dependent on the input type (for example if the left mouse button is clicked then it sets a Boolean condition of LeftMouseDown to true). The NEA_Physics_Engine.Utilities namespace contains the mathsUtility which contains a number of mathematical formulae which are used throughout the program.
The other two namespaces on this level break down even further. NEA_Physics_Engine.Physics contains the PhysicsManager which applies the physics to each object and manages the list of objects which have physics, the PhysicsBody which deals with the displacement, velocity and acceleration of the object, the PhysicsShape which sets up an objects collider, PhysicShapeCircle which sets up a circle's physical properties and PhysicsShapePolygon which sets up a polygon's physical properties. It also contains the namespace

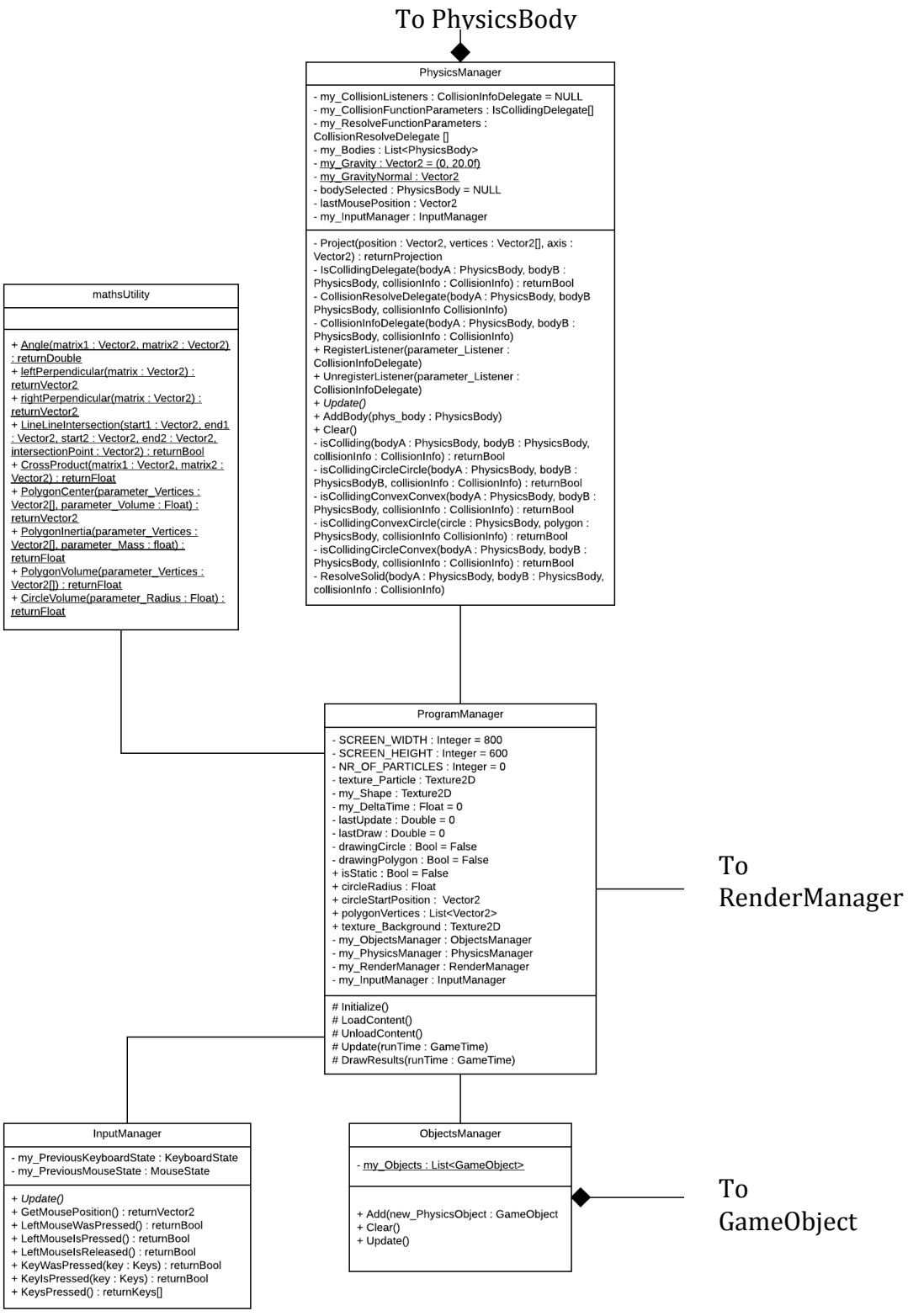
NEA_Physics_Engine.Physics.Properties which contains PhysicsProperties which contains all the constraints on values such as maximum velocity and material. NEA_Physics_Engine.Rendering contains the RenderManager which queues and renders the objects and the namespace NEA_Physics_Engine.Rendering.Render. This namespace contains RenderObject which sets the texture of the object, RenderCircle which sets the rendering properties of a circle and RenderLine which sets the rendering properties of the edge of a polygon.


NEA_Physics_Engine
Program
ProgramManager
ObjectsManager
GameObject
NEA_Physics_Engine.Physics
NEA_Physics_Engine.Rendering
NEA_Physics_Engine.Input
NEA_Physics_Engine.Utilities
NEA_Physics_Engine.Interfaces
NEA_Physics_Engine.Interfaces
Updateable
NEA_Physics_Engine.Physics
PhysicsManager
PhysicsBody
PhysicsShape
PhysicsShapeCircle
PhysicsShapePolygon
NEA_Physics_Engine.Physics.Properties
NEA_Physics_Engine.Rendering
RenderManager
NEA_Physics_Engine.Rendering.Render
NEA_Physics_Engine.Input
InputManager
NEA_Physics_Engine.Utilities
mathsUtility
NEA_Physics_Engine.Physics.Properties
PhysicsProperties
NEA_Physics_Engine.Rendering.Render
RenderObject
RenderLine
RenderCircle
Key
Namespace
Namespace.Subdivision
File

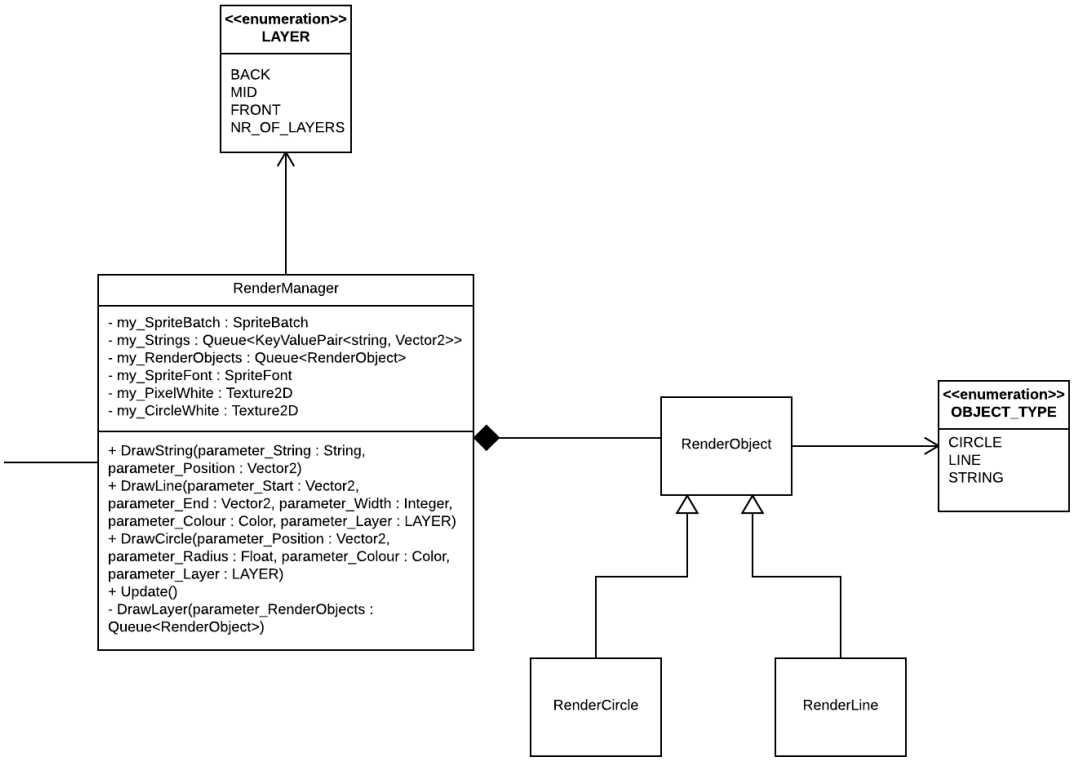
There are many files within each namespace, which will interact with one another in order to create the engine itself. The way in which these are related is more complex than the simple namespace relation, as each file can have multiple instances and thus can occur multiple times for a single instance of another file. The way in which these instances are related is given below.
The Program Manager displays the program on the Updateable interface and calls the Maths Utility to perform vector calculations, it also calls the input manager to detect any relevant inputs. On top of this it calls the Render Manager to render new and updated objects, the Objects Manager to update the list of objects when a new object is created or removed and the Physics Manager to deal with the physics of each object.
The Render Manager has many instances of Render Object as many objects need to be rendered and each Render Object is either a line or a circle so Render Object calls Render Line or Render Circle respectively.
The Objects Manager stores a list of objects each of which is an instance of Game Object.
The Physics Manager needs to know the constraints on the physics it's applying so calls Physics Properties. It also calls the Maths Utility for vector and volume calculations. Every object the Physics Manager affects is an instance of Physics Body which has a collider from Physics Shape and the collider is determined by its shape so Physics Shape calls the corresponding file.



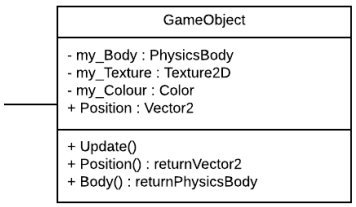
Now that the interactions between classes have been shown on a basic level, the classes need to be shown in detail. The diagram below is a class diagram illustrating the methods and attributes of each class and how they are related.


To PhysicsBody
PhysicsManager
- my_CollisionListeners : CollisionInfoDelegate = NULL
- my_CollisionFunctionParameters : IsCollidingDelegate[]
- my_ResolveFunctionParameters : CollisionResolveDelegate []
- my_Bodies : List<PhysicsBody>
- my_Gravity : Vector2 = (0, 20.0f)
- my_GravityNormal : Vector2
- bodySelected : PhysicsBody = NULL
- lastMousePosition : Vector2
- my_InputManager : InputManager
- Project(position : Vector2, vertices : Vector2[], axis : Vector2) : returnProjection
- IsCollidingDelegate(bodyA : PhysicsBody, bodyB : PhysicsBody, collisionInfo : CollisionInfo) : returnBool
- CollisionResolveDelegate(bodyA : PhysicsBody, bodyB PhysicsBody, collisionInfo CollisionInfo)
- CollisionInfoDelegate(bodyA : PhysicsBody, bodyB : PhysicsBody, collisionInfo : CollisionInfo)
+ RegisterListener(parameter_Listener : CollisionInfoDelegate)
+ UnregisterListener(parameter_Listener : CollisionInfoDelegate)
+ Update()
+ AddBody(phys_body : PhysicsBody)
+ Clear()
- isColliding(bodyA : PhysicsBody, bodyB : PhysicsBody, collisionInfo : CollisionInfo) : returnBool
- isCollidingCircleCircle(bodyA : PhysicsBody, bodyB : PhysicsBodyB, collisionInfo : CollisionInfo) : returnBool
- isCollidingConvexConvex(bodyA : PhysicsBody, bodyB : PhysicsBody, collisionInfo : CollisionInfo) : returnBool
- isCollidingConvexCircle(circle : PhysicsBody, polygon : PhysicsBody, collisionInfo CollisionInfo) : returnBool
- isCollidingCircleConvex(bodyA : PhysicsBody, bodyB : PhysicsBody, collisionInfo : CollisionInfo) : returnBool
- ResolveSolid(bodyA : PhysicsBody, bodyB : PhysicsBody, collisionInfo : CollisionInfo)
mathsUtility
+ Angle(matrix1 : Vector2, matrix2 : Vector2) : returnDouble
+ leftPerpendicular(matrix : Vector2) : returnVector2
+ rightPerpendicular(matrix : Vector2) : returnVector2
+ LineLineIntersection(start1 : Vector2, end1 : Vector2, start2 : Vector2, end2 : Vector2, intersectionPoint : Vector2) : returnBool
+ CrossProduct(matrix1 : Vector2, matrix2 : Vector2) : returnFloat
+ PolygonCenter(parameter_Vertices : Vector2[], parameter_Volume : Float) : returnVector2
+ PolygonInertia(parameter_Vertices : Vector2[], parameter_Mass : float) : returnFloat
+ PolygonVolume(parameter_Vertices : Vector2[]) : returnFloat
+ CircleVolume(parameter_Radius : Float) : returnFloat
ProgramManager
- SCREEN_WIDTH : Integer = 800
- SCREEN_HEIGHT : Integer = 600
- NR_OF_PARTICLES : Integer = 0
- texture_Particle : Texture2D
- my_Shape : Texture2D
- my_DeltaTime : Float = 0
- lastUpdate : Double = 0
- lastDraw : Double = 0
- drawingCircle : Bool = False
- drawingPolygon : Bool = False
+ isStatic : Bool = False
+ circleRadius : Float
+ circleStartPosition : Vector2
+ polygonVertices : List<Vector2>
+ texture_Background : Texture2D
- my_ObjectsManager : ObjectsManager
- my_PhysicsManager : PhysicsManager
- my_RenderManager : RenderManager
- my_InputManager : InputManager
Initialize()
LoadContent()
UnloadContent()
Update(runTime : GameTime)
DrawResults(runTime : GameTime)
To RenderManager
InputManager
- my_PreviousKeyboardState : KeyboardState
- my_PreviousMouseState : MouseState
+ Update()
+ GetMousePosition() : returnVector2
+ LeftMouseWasPressed() : returnBool
+ LeftMouseIsPressed() : returnBool
+ LeftMouseIsReleased() : returnBool
+ KeyWasPressed(key : Keys) : returnBool
+ KeyIsPressed(key : Keys) : returnBool
+ KeysPressed() : returnKeys[]
ObjectsManager
- my_Objects : List<GameObject>
+ Add(new_PhysicsObject : GameObject
+ Clear()
+ Update()
To GameObject

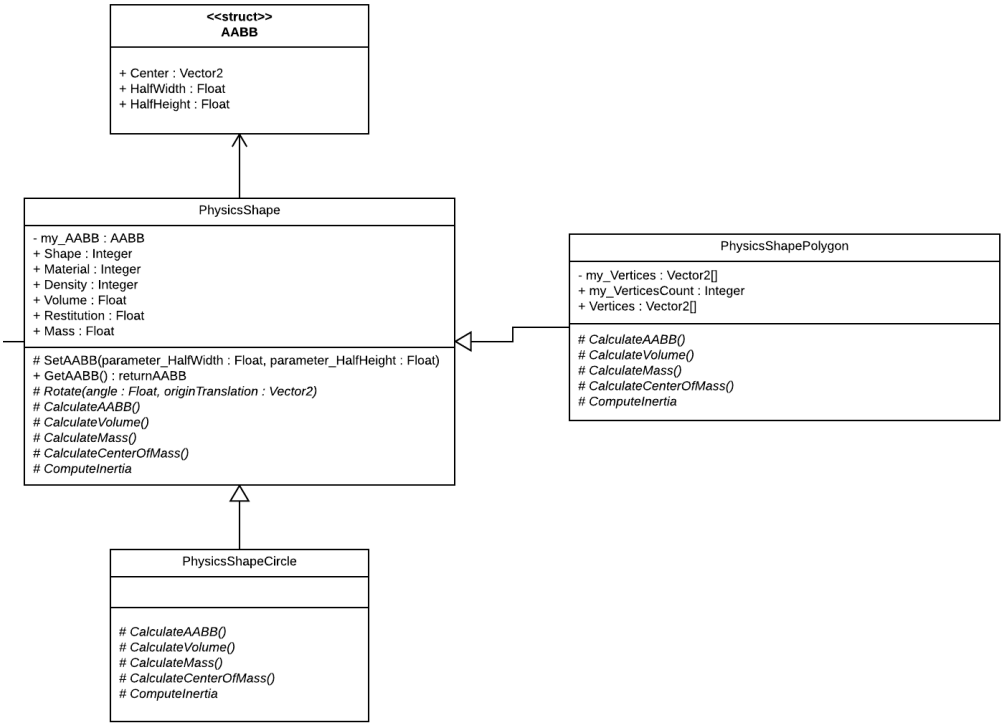
To ProgramManager

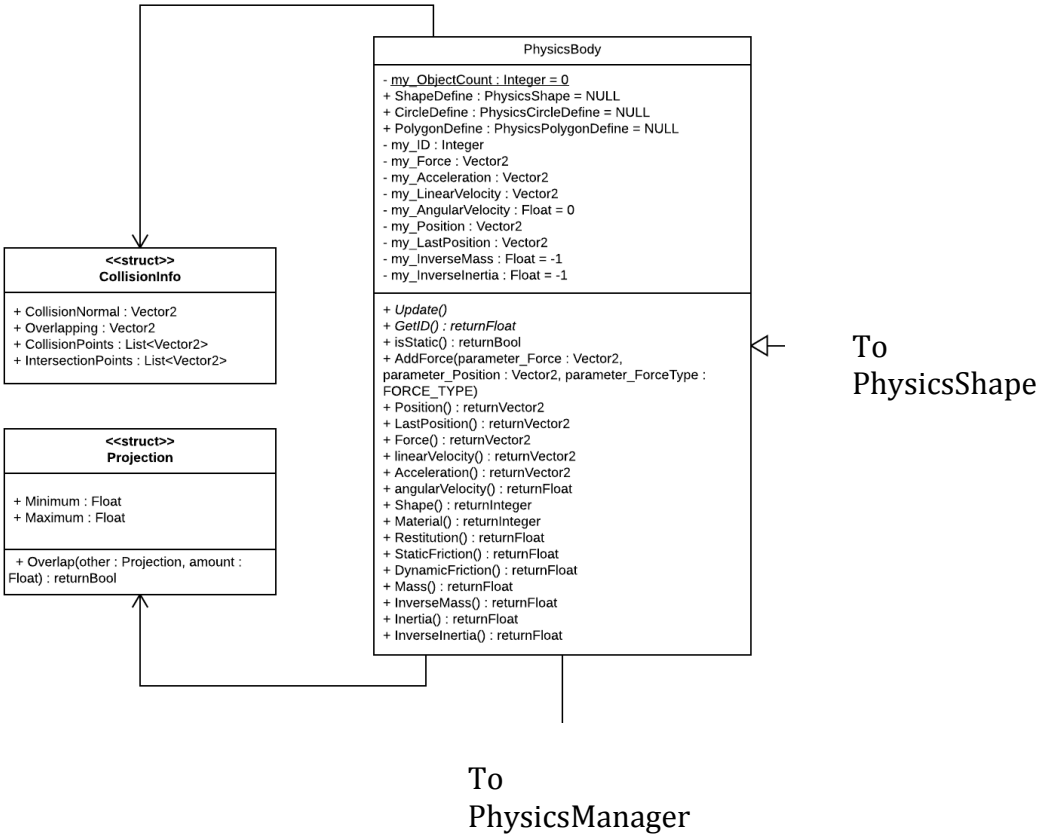


To ObjectsManager



To PhysicsBody




PhysicsBody
- my_ObjectCount : Integer = 0
+ ShapeDefine : PhysicsShape = NULL
+ CircleDefine : PhysicsCircleDefine = NULL
+ PolygonDefine : PhysicsPolygonDefine = NULL
- my_ID : Integer
- my_Force : Vector2
- my_Acceleration : Vector2
- my_LinearVelocity : Vector2
- my_AngularVelocity : Float = 0
- my_Position : Vector2
- my_LastPosition : Vector2
- my_InverseMass : Float = -1
- my_InverseInertia : Float = -1
+ Update()
+ GetID() : returnFloat
+ isStatic() : returnBool
+ AddForce(parameter_Force : Vector2, parameter_Position : Vector2, parameter_ForceType : FORCE_TYPE)
+ Position() : returnVector2
+ LastPosition() : returnVector2
+ Force() : returnVector2
+ linearVelocity() : returnVector2
+ Acceleration() : returnVector2
+ angularVelocity() : returnFloat
+ Shape() : returnInteger
+ Material() : returnInteger
+ Restitution() : returnFloat
+ StaticFriction() : returnFloat
+ DynamicFriction() : returnFloat
+ Mass() : returnFloat
+ InverseMass() : returnFloat
+ Inertia() : returnFloat
+ InverseInertia() : returnFloat
<<struct>>
CollisionInfo
+ CollisionNormal : Vector2
+ Overlapping : Vector2
+ CollisionPoints : List<Vector2>
+ IntersectionPoints : List<Vector2>
<<struct>>
Projection
+ Minimum : Float
+ Maximum : Float
+ Overlap(other : Projection, amount : Float) : returnBool
To PhysicsShape
To PhysicsManager

Pseudo Code

Program Manager

```
CLASS ProgramManager
    SET Screen Properties
    NR_OF_PARTICLES = 0
    INITIALISE Textures
    deltaTime = 0

    METHOD Initialize
        my_PhysicsManager = Instance of PhysicsManager
        my_RenderManager = Instance of RenderManager
        my_InputManager = Instance of InputManager
        my_ObjectsManager = Instance of ObjectsManager
    ENDMETHOD
    METHOD LoadContent
        LOAD TEXTURES
        My_ObjectsManager.Add (Scene Shapes)
    ENDMETHOD
    METHOD drawPreset1
        My_ObjectsManager.Add (Scene shapes)
    ENDMETHOD
    METHOD drawPreset2
        My_ObjectsManager.Add (Scene shapes)
    ENDMETHOD
    METHOD drawPreset3
        My_ObjectsManager.Add (Scene shapes)
    ENDMETHOD
    METHOD drawPreset4
        My_ObjectsManager.Add (Scene shapes)
    ENDMETHOD
    METHOD drawPreset5
        My_ObjectsManager.Add (Scene shapes)
    ENDMETHOD
    METHOD drawPreset6
        My_ObjectsManager.Add (Scene shapes)
    ENDMETHOD
    METHOD drawPreset7
        My_ObjectsManager.Add (Scene shapes)
    ENDMETHOD

    METHOD UnloadContent
        UNLOAD
    ENDMETHOD

    drawingCircle = false
    drawingPolygon = false
    preset = 0
    isStatic = false
    showControls = true
```

```
INITIALISE Possible Shape Properties (vertices, radius, etc.)

METHOD Update
        deltaTime = Time program has been running
        IF (my_InputManager.KeyWasPressed(ESC)) THEN
                EXIT PROGRAM
        ENDIF

        IF (my_InputManager.KeyWasPressed(1)) THEN
                preset = 1
        ENDIF
        IF (my_InputManager.KeyWasPressed(2)) THEN
                preset = 2
        ENDIF
        IF (my_InputManager.KeyWasPressed(3)) THEN
                preset = 3
        ENDIF
        IF (my_InputManager.KeyWasPressed(4)) THEN
                preset = 4
        ENDIF
        IF (my_InputManager.KeyWasPressed(5)) THEN
                preset = 5
        ENDIF
        IF (my_InputManager.KeyWasPressed(6)) THEN
                preset = 6
        ENDIF
        IF (my_InputManager.KeyWasPressed(7)) THEN
                preset = 7
        ENDIF
        IF (my_InputManager.KeyWasPressed(0)) THEN
                preset = 0
        ENDIF
        IF (my_InputManager.KeyWasPressed(C)) THEN
                My_ObjectsManager.Clear
        ENDIF
        IF (my_InputManager.KeyWasPressed(Q)) THEN
                IF (isStatic) THEN
                        isStatic = false
                 ELSE
                        isStatic = true
                ENDIF
        ENDIF
        IF (drawingPolygon = true OR drawingCircle = true) THEN
                RenderManager.DrawString ("Static: " + isStatic)
                RenderManager.DrawString ("Material: " + Material)
        ENDIF
        IF (my_InputManager.KeyWasPressed(P)) THEN
                IF (drawingPolygon = false) THEN
                        drawingPolygon = true
```

```
                ELSE IF ( polygonVertices.length > 2)  THEN
                        INITIALISE startPosition
                        FOR EACH (vertex IN polygonVertices)
                                startPosition = startPosition + vertex
                        ENDFOR EACH
                        startPosition.X = startPosition.X /
polygonVertices.Count
                        startPosition.Y = startPosition.Y/
polygonVertices.Count
                        my_ObjectsManager.Add(Properties)
                        polygonVertices.Clear
                        drawingPolygon = false

                ELSE
                        drawingPolygon = false
                        polygonVertices.Clear
                ENDIF
        ELSEIF (my_InputManager.KeyWasPressed(O)) THEN
                IF (drawingCircle = false) THEN
                        drawingCircle = true
                ELSEIF (circleStartPosition != 0) THEN
                        My_ObjectsManager.Add(Properties)
                        circleStartPosition = 0
                        circleRadius = 0
                        drawingCircle = false
                ELSE
                        drawingCircle = false
                ENDIF
        ELSEIF (my_InputManager.KeyWasPressed(X)) THEN
                IF (showControls = false) THEN
                        showControls = true
                ELSE
                        showControls = false
                ENDIF
        ENDIF
        IF (drawingPolygon) THEN
                RenderManager.DrawString("Drawing Polygon")
                FOR EACH (vertex IN polygonVertices)
                        RenderManager.DrawCircle(vertex, 6)
                ENDFOR EACH
                IF (polygonVertices.Count > 2) THEN
                        normal1 =
                        mathsUtility.LeftPerpendicular(polygonVertices[pol
                        ygonVertices.Count - 1] -
                        polygonVertices[polygonVertices.Count - 2])

                        normal2 =
                        mathsUtility.LeftPerpendicular(polygonVertices[1] -
                        polygonVertices[0])
```

```
                normal3 =
                mathsUtility.LeftPerpendicular(polygonVertices[0] -
                polygonVertices[polygonVertices.Count - 1])
                newEdge1 = polygonVertices[polygonVertices.Count
                - 1] - my_InputManager.GetMousePosition()

                newEdge2 = polygonVertices[0] -
                my_InputManager.GetMousePosition()

                IF (newEdge1.normal1 < 0 AND newEdge2.normal2
                < 0 AND newEdge3.normal3 > 0) THEN

                        RenderManager.DrawCircle(my_InputManag
                        er.GetMousePosition , 6, Green)

                        IF (my_InputManager.LeftMouseWasPressed)
                        THEN
                                polygonVertices.Add(my_InputManag
                                er.GetMousePosition)
                        ENDIF
                ELSE
                        RenderManager.DrawCircle(my_InputManag
                        er.GetMousePosition , 6, Red)
                ENDIF
        ELSE
                IF (polygonVertices = 2) THEN
                        edge = my_InputManager.GetMousePosition –
                        polygonVertices[1]

                        normal =
                        mathsUtility.LeftPerpendicular(polygonVerti
                        ces[1] - polygonVertices[0])

                        IF (edge.normal > 0) THEN
                                RenderManager.DrawCircle(my_Input
                                Manager.GetMousePosition , 6, Green)
                                IF
                        (my_InputManager.LeftMouseWasPressed)
                                        polygonVertices.Add(my_Input
                                        Manager.GetMousePosition)
                                        THEN
                                ENDIF
                            ELSE

RenderManager.DrawCircle(my_InputManag er.Ge tMousePosition, 6,
                                                        Green)
                        ENDIF
                ELSE
```

```
                                        RenderManager.DrawCircle(my_InputManag
                                        er.GetMousePosition , 6, Green)
                                        IF (my_InputManager.LeftMouseWasPressed)
                                                polygonVertices.Add(my_InputManag
                                                er.GetMousePosition)
                                        ENDIF
                                ENDIF
                        ENDIF
                ELSEIF (drawingCircle) THEN
                        RenderManager.DrawString("Drawing Circle")
                        IF (my_InputManager.LeftMouseWasPressed) THEN
                                circleStartPosition =
my_InputManager.GetMousePosition
                        ENDIF
                        IF (my_InputManager.LeftMouseIsPressed) THEN
                                circleRadius = (circleStartPosition –
                                my_InputManager.GetMousePosition).Length
                        ENDIF
                        RenderManager.DrawCircle(circleStartPosition,
circleRadius, Green)
                ENDIF
                IF (showControls = true) THEN
                        >> Write the controls <<
                ENDIF
                my_PhysicsManager.Update
                my_ObjectsManager.Update
                my_InputManager.Update
                my_RenderManager.Update
        ENDMETHOD

        GET deltaTime
        GET Screen Properties

        INSTANTIATE ProgramManager
ENDCLASS
```

Render Manager

```
CLASS RenderManager
        NR_OF_LAYERS = 3
        my_Strings = EMPTY QUEUE
        my_RenderObjects[] = EMPTY QUEUE

        LOAD TEXTURES

        METHOD DrawString
                Add given string to end of my_Strings
        ENDMETHOD

        METHOD DrawLine
                Add given line to end of my_RenderObjects[0]
        ENDMETHOD

        METHOD DrawCircle
                Add given circle to end of my_RenderObjects[0]
        ENDMETHOD

        METHOD UPDATE
                FOR (i = 1 TO NR_OF_LAYERS)
                        DrawLayer(my_RenderObjects[i])
                ENDFOR
                FOR EACH (String IN my_Strings)
                        DrawString
                ENDFOR EACH
                FOR (i = 1 TO NR_OF_LAYERS)
                        my_RenderObjects.Clear
                ENDFOR
                My_Strings.Clear
        ENDMETHOD

        METHOD DrawLayer
                FOR EACH (renderObject IN parameter_RenderObjects)
                        CASE renderObject.Type OF
                                Line: line = renderObject
                                      lineVector = line.End – line.Start
                                      lineLength = lineVector.Length
                                      perpendicularDotProduct = lineVector.X *
        UnitX.Y -
                                      lineVector.Y * UnitX.X
                                      DRAW LINE
                                Circle: circle = renderObject
                                        DRAW CIRCLE
                        ENDCASE
                ENDFOR EACH
        ENDMETHOD
```

```
        INSTANTIATE RenderManager
ENDCLASS
```

Render Object

```
ENUM OBJECT_TYPE
        CIRCLE,
        LINE,
        STRING
ENDENUM

CLASS RenderObject
        Type = GET OBJECT_TYPE
        Colour = GET Color

        STRUCTURE RenderObject
                Type = parameter_Type
                IF  (parameter_Colour.A != 0)
                        Colour = parameter_Colour;
                ELSE
                        Colour = Color.White;
        ENDSTRUCTURE
ENDCLASS
```

Input Manager

```
CLASS InputManager
        my_PreviousKeyboardState = NULL
        my_PreviousMouseState = NULL
        METHOD Update
                my_PreviousKeyboardState = Keyboard.GetState
                my_PreviousMouseState = Mouse.GetState
        ENDMETHOD
        METHOD GetMousePosition
                RETURN (Mouse.GetState.X, Mouse.GetState.Y)
        ENDMETHOD
        METHOD LeftMouseWasPressed
                RETURN ((Mouse.GetState.LeftButton = ButtonState.Pressed) &&
                (my_PreviousMouseState.LeftButton != ButtonState.Pressed))
        ENDMETHOD
        METHOD LeftMouseIsPressed
                RETURN (Mouse.GetState.LeftButton = ButtonState.Pressed)
        ENDMETHOD
        METHOD KeyWasPressed
                return Keyboard.GetState.IsKeyDown(key) &&
                my_PreviousKeyboardState.IsKeyUp(key)
        ENDMETHOD
        METHOD KeysPressed
                return Keyboard.GetState.GetPressedKeys
        ENDMETHOD
        INSTANTIATE InputManager
ENDCLASS
```

Maths Utility

```
CLASS mathsUtility
        METHOD Angle
                NORMALIZE parameter_Vectors
                CALCULATE Angle
                IF (Angle < 0.0001)
                        RETURN 0
                ELSE
                        RETURN Angle
                ENDIF
        ENDMETHOD
        METHOD leftPerpendicular
                RETURN (-parameter_Vector.Y, parameter_Vector.X)
        ENDMETHOD
        METHOD rightPerpendicular
                RETURN (parameter_Vector.Y, -parameter_Vector.X)
        ENDMETHOD
        METHOD CrossProduct
                RETURN (matrix1.X * matrix2.Y) - (matrix1.Y * matrix2.X)
        ENDMETHOD
        METHOD PolygonCenter
                center = NULL
                FOR i = 0 TO parameter_Vertices.Length DO
                        vertex = parameter_Vertices[i]
                        center.X += (vertex1.X + vertex2.X) * (vertex1.X * vertex2.Y
                        - vertex2.X     * vertex1.Y)
                        center.Y += (vertex1.Y + vertex2.Y) * (vertex1.X * vertex2.Y
                        - vertex2.X * vertex1.Y)
                ENDFOR
                center.X *= 1/ (6 * parameter_Volume)
                center.Y *= 1/ (6 * parameter_Volume)
                RETURN center
        ENDMETHOD
        METHOD PolygonInertia
                sum1 = 0
                sum2 = 0
                FOR ( i = 0 TO parameter_Vertices.Length) DO
                        v1 = parameter_Vertices[i]
                        v2 = parameter_Vertices[(i + 1)  MOD
parameter_Vertices.Length]
                        v2.Normalize()
                        a = Vector2.Dot(v2, v1)
                        b = Vector2.Dot(v1, v1) + Vector2.Dot(v1, v2) +
Vector2.Dot(v2, v2)

                         sum1 += a * b
                        sum2 += a
                ENDFOR
                RETURN Math.Abs((parameter_Mass * sum1) / (6.0f * sum2))
```

```
        ENDMETHOD
        METHOD PolygonVolume
                area = 0
                j = parameter_Vertices.Length – 1
                FOR i = 0 TO parameter_Vertices.Length DO
                        area += (parameter_Vertices[j].X + parameter_Vertices[i].X)
                        * (parameter_Vertices[j].Y - parameter_Vertices[i].Y)
                        j = i
                ENDFOR
                RETURN |area * 0.5|
        ENDMETHOD
        METHOD CircleVolume
                area = π * (parameter_Radius * parameter_Radius)
                RETURN |area * 0.5|
        ENDMETHOD
ENDCLASS
```

Objects Manager

```
CLASS ObjectsManager
        my_Objects = LIST OF GameObject
        METHOD Add
                ADD new_PhysicsObject TO my_Objects
        ENDMETHOD
        METHOD Clear
                CLEAR my_Objects
                CLEAR INSTANCE OF PhysicsManager
        ENDMETHOD
        METHOD Update
                FOREACH (GameObject Shape IN my_Objects)
                        UPDATE shape
                ENDFOREACH
        ENDMETHOD
        INSTANTIATE ObjectsManager
ENDCLASS
```

Game Object

```
CLASS GameObject
        my_Body = NEW PhysicsBody
        my_Texture = NEW Texture
        my_Colour = NEW Color

        METHOD Position
                RETURN my_Body.Position
        ENDMETHOD
        METHOD Update
                IF  Body.Shape = CIRCLE THEN
                        DRAW CIRCLE
                ELSE
                        FOR i = 0 TO Body.PolygonDefine.VerticesCount DO
                                DRAW LINE AT i
                        ENDFOR
                ENDIF
        ENDMETHOD
ENDCLASS
```

Physics Manager

```
CLASS PhysicsManager
    STRUCTURE Projection
        Minimum = minimum
        Maximum = maximum
        BOOL Overlap
            IF (Maximum >= other.Minimum AND other.Maximum >=
            Minimum) THEN
                IF (Maximum < other.Maximum) THEN
                    amount = |Maximum – other.Minimum|
                ELSE
                    amount = |other.Maximum – Minimum|
                ENDIF
                RETURN true
            ELSE
                amount = -1
                RETURN false
            ENDBOOL
    ENDSTRUCTURE
    PROJECTION Project
        minimum = (position + vertices[0]).axis
        maximum = minimum
        FOR (i = 0 to vertices.Length) DO
            p =  (position + vertices[i]).axis
            IF (p < minimum) THEN
                minimum = p
            ELSE IF (p > maximum)
                maximum = p
            ENDIF
        ENDFOR
        RETURN PROJECTION(minimum, maximum)
    ENDPROJECTION
    STRUCTURE CollisionInfo
        CollisionNormal (Vector2)
        Overlapping (Vector2)
        CollisionPoints (List)
        IntersectionPoints (List)
    ENDSTRUCTURE
    my_Bodies (List)
    my_Gravity = (0, 20.0f)
    my_GravityNormal = mathsUtility.leftPerpendicular(my_Gravity)
    my_Iterations = 3
    bodySelected = NULL
    lastMousePosition (Vector2)
    my_InputManager = InputManager.Instance
    METHOD RegisterListener
        my_CollisionListeners += parameter_Listener
    ENDMETHOD
    METHOD UnregisterListener
```

```
            my_CollisionListeners -= parameter_Listener
        ENDMETHOD
        METHOD Update
            collisionInfo (CollisionInfo)
            mousePosition = my_InputManager.GetMousePosition
            IF (bodySelected != NULL) THEN
                IF (|mousePosition.X – bodySelected.Position.X| < 10 AND
                |mousePosition.Y – bodySelected.Position.Y| < 10) THEN
                    ADD FORCE
                ENDIF
                ADD FORCE
            ENDIF
            FOREACH (bodyA IN my_Bodies) DO
                IF (bodyA.isStatic = FALSE) THEN
                    FOREACH (bodyB in my_Bodies)
                        IF (bodyA != bodyB) THEN
                            Resolve possible collision
                        ENDIF
                    ENDFOREACH
                ADD GRAVITY
                bodyA.Update
            ENDFOREACH
            lastMousePosition = mousePosition
        ENDMETHOD
        METHOD AddBody
            my_Bodies.Add(phys_body)
        ENDMETHOD
        METHOD Clear
            my_Bodies.Clear
        ENDMETHOD
        BOOL isColliding
            RETURN collisionInfo
        ENDBOOL
        BOOL isCollidingCircleCircle
            collisionInfo.CollisionPoints (List)
            collisionInfo.IntersectionPoints (List)
            distance = bodyA.Position – bodyB.Position
            widths = bodyA.CircleDefine.Radius + bodyB.CircleDefine.Radius

            IF (distance^2 < widths^2) THEN
                IF (distance.X = 0 AND distance.Y = 0) THEN
                    specialDistance = bodyA.LastPosition –
bodyB.Position
                    IF (specialDistance.X = 0 AND specialDistance.Y = 0)
THEN
                        specialDistance.X = 1
                    ELSE
                        specialDistance.Normalize
                    ENDIF
```

```
                        collisionInfo.Overlapping = specialDistance * widths
                        collisionInfo.CollisionNormal = specialDistance
                        collisionInfo.CollisionPoints (List)
                ELSE
                        overlappingAmount = widths – distance.Length
                        distance.Normalize
                        collisionInfo.Overlapping = distance *
overlappingAmount
                        collisionInfo.CollisionNormal = distance
                        collisionInfo.CollisionPoints.Add(bodyB.Position +
        (collisionInfor.CollisionNormal * bodyB.CircleDefine.Radius))
                ENDIF
                RETURN TRUE
            ELSE
                collisionInfo.CollisionNormal (Vector2)
                collisionInfo.Overlapping (Vector2)
                RETURN FALSE
            ENDIF
        ENDBOOL
        BOOL isCollidingConvexConvex
            collisionInfo.CollisionNormal (Vector2)
            collisionInfo.Overlapping (Vector2)
            collisionInfo.CollisionPoints (List)
            collisionInfo.IntersectionPoints (List)
            verticesInside (List)
            edge (Vector2)
            edgeNormal (Vector2)
            shortestOverlapAxis = bodyA.PolygonDefine.GetVertex(0)
            projectionA (Projection)
            projectionB (Projection)
            shortestOverlapAmount = MAXIMUM POSSIBLE INTEGER
            overlapAmount = 0
            FOR (i = 0 TO bodyA.PolygonDefine.VerticesCount) DO
                edge = bodyA.PolygonDefine.GetVertex(i+1) –
bodyA.PolygonDefine.GetVertex(i)
                edgeNormal = mathsUtility.RightPerpendicular(edge)
                edgeNormal.Normalize
                projectionA = Project(bodyA.Position,
bodyA.PolygonDefine.Vertices, edgeNormal)
                projectionB = Project(bodyB.Position,
bodyA.PolygonDefine.Vertices, edgeNormal)
                IF (ProjectionA DOESN'T Overlap with ProjectionB) THEN
                    return FALSE
                ELSE IF (overlapAmount < shortestOverlapAmount) THEN
                    shortestOverlapAxis = edgeNormal
                    shortestOverlapAmount = overlapAmount
                ENDIF
                FOR (j = 0 TO bodyB.PolygonDefine.VerticesCount) DO
```

```
                        point1 = (bodyB.PolygonDefine.GetVertex(j) +
bodyB.Position)
                        point2 = (bodyB.PolygonDefine.GetVertex(j+1) +
bodyB.Position)
                        point3 = (bodyA.PolygonDefine.GetVertex(i) +
bodyA.Position)
                        point4 = (bodyA.PolygonDefine.GetVertex(i+1) +
bodyA.Position)
                        intersectionPoint (Vector2)
                        IF (mathsUtility.LineLineIntersection(point1, point2,
point3, point4, OUT intersectionPoint)) THEN

        collisionInfo.IntersectionPoints.Add(intersectionPoint)
                    ENDFOR
                    IF (i = 0) THEN
                        FOR (j = 0 TO bodyB.PolygonDefine.VerticesCount)
DO
                            vertex = (bodyB.PolygonDefine.GetVertex(j) +
bodyB.Position) - (bodyA.PolygonDefine.GetVertex(i) + bodyA.Position)
                            dot = vertex.edgeNormal
                            IF (dot < 0) THEN

verticesInside.Add(bodyB.PolygonDefine.GetVertex(j) + bodyB.Position)
                            ENDIF
                        ENDFOR
                    ELSE
                        FOR (j = 0 TO verticesInside.Count) DO
                            vertex = verticesInside[j] -
(bodyA.PolygonDefine.GetVertex(i) + bodyA.Position)
                            dot = vertex.edgeNormal
                            IF (dot > 0) THEN
                                verticesInside.RemoveAt(j)
                                j--
                            ENDIF
                        ENDFOR
                    ENDIF
                ENDFOR
                FOR (i = 0 TO bodyB.PolygonDefine.VerticesCount) DO
                    edge = bodyB.PolygonDefine.GetVertex(i+1) –
bodyB.PolygonDefine.GetVertex(i)
                    edgeNormal = mathsUtility.RightPerpendicular(edge)
                    edgeNormal.Normalize
                    projectionB = Project(bodyA.Position,
bodyA.PolygonDefine.Vertices, edgeNormal)
                    projectionA = Project(bodyB.Position,
bodyB.PolygonDefine.Vertices, edgeNormal)
                    IF (ProjectionA DOESN'T Overlap with ProjectionB) THEN
                        RETURN FALSE
                    ELSE IF (overlapAmount < shortestOverlapAmount) THEN
```

```
                        shortestOverlapAxis = edgeNormal
                        shortestOverlapAmount = overlapAmount
                    ENDIF
                    IF (i = 0) THEN
                        FOR (j = 0 TO bodyA.PolygonDefine.VerticesCount)
DO
                            vertex = (bodyA.PolygonDefine.GetVertex(j) +
bodyA.Position) - (bodyB.PolygonDefine.GetVertex(i) + bodyB.Position)
                            dot = vertex.edgeNormal
                            IF (dot < 0) THEN
                                verticesInside.Add
(bodyA.PolygonDefine.GetVertex(j) + bodyA.Position)
                            ENDIF
                        ENDFOR
                    ELSE
                        FOR (j = 0 to VerticesInside.Count) DO
                            vertex = verticesInside[j] -
(bodyB.PolygonDefine.GetVertex(i) + bodyB.Position)
                            dot = vertex.edgeNormal
                            IF (dot > 0) THEN
                                verticesInside.RemoveAt(j)
                                j--
                            ENDIF
                        ENDFOR
                    ENDIF
                ENDFOR
                collisionInfo.CollisionNormal = shortestOverlapAxis
                collisionInfo.Overlapping = shortestOverlapAxis *
shortestOverlapAmount
                collisionInfo.CollisionPoints.AddRange(verticesInside)
                RETURN TRUE
            ENDBOOL
            BOOL IsCollidingConvexCircle
                collisionInfo.CollisionPoints (List)
                collisionInfo.IntersectionPoints (List)
                closestProjection = (ProgramManager.ScreenWidth,
ProgramManager.ScreenHeight)
                insidePolygon = TRUE
                FOR (i = 0 TO polygon.PolygonDefine.VerticesCount) DO
                    vectorToCircle = circle.Position – (polygon.Position +
polygon.PolygonDefine.GetVertex(i)
                    currentEdge = polygon.PolygonDefine.GetVertex(i+1) –
polygon.PolygonDefine.GetVertex(i)
                    currentEdgeLengthSquared = currentEdge.LengthSquared
                    currentEdge.Normalize
                    circleToEdgeProjection = vectorToCircle.currentEdge
                    IF (circleToEdgeProjection > 0) THEN
                        IF (circleToEdgeProjection^2 <
currentEdgeLengthSquared) THEN
```

```
                                        vectorCircleProjectionEdge = currentEdge *
circleToEdgeProjection
                                        projectionToCircle = vectorToCircle –
vectorCircleProjectionEdge
                                        projectionToCircleLengthSquared =
projectionToCircle.LengthSquared
                                        IF (projectionToCircleLengthSquared <
closestProjection.LengthSquared) THEN
                                                closestProjection = projectionToCircle
                                        ENDIF
                                        IF (projectionToCircleLengthSquared <
circle.CircleDefine.Radius^2) THEN
                                                collisionInfo.CollisionNormal = -
projectionToCircle

        collisionInfo.CollisionNormal.Normalize

        collisionInfo.CollisionPoints.Add(circle.Position +
(collisionInfo.CollisionNormal * circle.CircleDefine.Radius))
                                                collisionInfo.Overlapping =
collisionInfo.CollisionNormal * (circle.CircleDefine.Radius -
projectionToCircle.Length)
                                                RETURN TRUE
                                        ENDIF
                                ENDIF
                        ELSE IF (vectorToCircle.LengthSquared <
circle.CircleDefine.Radius^2) THEN
                                collisionInfo.CollisionNormal = - vectorToCircle
                                collisionInfo.CollisionNormal.Normalize
                                collisionInfo.CollisionPoints.Add(
polygon.PolygonDefine.GetVertex(i) + polygon.Position)
                                collisionInfo.Overlapping =
collisionInfo.CollisionNormal * (circle.CircleDefine.Radius -
vectorToCircle.Length)
                                RETURN TRUE
                        ENDIF
                        edgeNormal =
mathsUtility.RightPerpendicular(currentEdge)
                        IF (vectorToCircle.edgeNormal > 0) THEN
                                insidePolygon = FALSE
                        ELSE IF (i = polygon.PolygonDefine.VerticesCount – 1)
THEN
                                collisionInfo.CollisionNormal = vectorToCircle
                                collisionInfo.CollisionNormal.Normalize
                                collisionInfo.Overlapping = vectorToCircle *
(vectorToCircle.Length() + circle.CircleDefine.Radius)
                                collisionInfo.CollisionPoints.Add(
polygon.PolygonDefine.GetVertex(0) + polygon.Position)
                                RETURN insidePolygon
```

```
                ENDIF
            ENDFOR
            collisionInfo.CollisionNormal (Vector2)
            collisionInfo.Overlapping (Vector2)
            RETURN FALSE
        ENDBOOL
        BOOL IsCollidingCircleConvex
            RETURN IsCollidingConvexCircle
        ENDBOOL
        METHOD ResolveSolid
            IF (BodyA.Position - BodyB.Position . collisionInfo.CollisionNormal
< 0) THEN
                collisionInfo.CollisionNormal = -
collisionInfo.CollisionNormal
                collisionInfo.Overlapping = -collisionInfo.Overlapping
            ENDIF
            bodyA.Position += collisionInfo.Overlapping
            j = 0
            collisionPointRelativeVelocityAtoB = 1 in x-direction
            FOREACH (collisionPoint IN collisionInfo.CollisionPoints) DO
                collisionPointRadiusA = collisionPoint - bodyA.Position
                collisionPointRadiusB = collisionPoint - bodyB.Position
                collisionPointRadiusAPerpendicular =
mathsUtility.rightPerpendicular(collisionPointRadiusA)
                collisionPointRadiusBPerpendicular =
mathsUtility.rightPerpendicular(collisionPointRadiusB)
                collisionPointVelocityA = bodyA.linearVelocity
                collisionPointVelocityB = bodyB.linearVelocity
                collisionPointRelativeVelocityAtoB =
collisionPointVelocityA – collisionPointVelocityB
                collisionTangent = collisionPointRelativeVelocityAtoB –
(collisionPointRelativeVelocityAtoB . collisionInfo.CollisionNormal) *
collisionInfo.CollisionNormal
                collisionTangent.Normalize()
                j = (-(1 + (bodyA.Restitution + bodyB.Restitution) * 0.5f) *
collisionPointRelativeVelocityAtoB  .  collisionInfo.CollisionNormal) /
collisionInfo.CollisionNormal . (collisionInfo.CollisionNormal *
(bodyA.InverseMass + bodyB.InverseMass))
                jt = -(1 + (bodyA.Restitution + bodyB.Restitution) * 0.5f )
(collisionPointRelativeVelocityAtoB . collisionTangent)/(bodyA.InverseMass +
bodyB.InverseMass)
                CoefficientOfFriction = (bodyA.StaticFriction +
bodyB.StaticFriction) * 0.5f
                IF (|jt| >= j * CoefficientOfFriction) THEN
                    CoefficientOfFriction = (bodyA.DynamicFriction +
bodyB.DynamicFriction) * 0.5f
                    jt = -j * CoefficientOfFriction
                ENDIF
```

```
                        IF (j * collisionInfo.CollisionNormal .
collisionInfo.CollisionNormal < 0) THEN
                                j = -j
                        ENDIF
                        BodyA.AddForce
                        BodyB.AddForce
                ENDFOREACH
        ENDMETHOD
        INSTANTIATE PhysicsManager
ENDCLASS
```

Physics Body

```
CLASS PhysicsBody
        my_ObjectCount = 0
        ShapeDefine = NULL
        PolygonDefine = NULL
        CircleDefine = NULL
        my_Force = NEW Vector2
        my_Acceleration = NEW Vector2
        my_LinearVelocity = NEW Vector2
        my_AngularVelocity = 0
        my_Position = NEW Vector2
        my_LastPosition = NEW Vector2
        my_InverseMass = -1
        my_InverseInertia = -1
        METHOD PhysicsBody
                ShapeDefine = parameter_ShapeDefine
                Position = parameter_Position
                Restitution = parameter_ShapeDefine.Restitution
                StaticFriction = parameter_ShapeDefine.StaticFriction
                DynamicFriction = parameter_ShapeDefine.DynamicFriction

                Mass = parameter_ShapeDefine.Mass;
                my_ID = my_ObjectCount++;

                IF (parameter_ShapeDefine.Shape == 1) THEN
                        CircleDefine =
(PhysicsCircleDefine)parameter_ShapeDefine
                ELSE IF (parameter_ShapeDefine.Shape == 2)
                        PolygonDefine =
(PhysicsPolygonDefine)parameter_ShapeDefine
                ENDIF
                PhysicsManager.AddBody()
        ENDMETHOD
        METHOD Update
                LastPosition = Position
                deltaTime = ProgramManager.DeltaTime
                ShapeDefine.Rotate(0.2f * angularVelocity * deltaTime)
                Acceleration = Force * InverseMass
                linearVelocity += Acceleration * deltaTime
                linearVelocity *= 0.99f
                angularVelocity *= 0.995f
                Position += (linearVelocity * deltaTime)
                IF (isStatic() == false) THEN
                        RenderManager.DrawCircle(Position, 6)
                ENDIF
                AddForce(-Force, Position)
        ENDMETHOD
        METHOD AddForce
                IF (parameter_ForceType == FORCE_TYPE.FORCE) THEN
```

```
                Force += parameter_Force
            ELSE IF (parameter_ForceType == FORCE_TYPE.IMPULSE) THEN
                linearVelocity += parameter_Force * InverseMass
            ENDIF
            radiusVector = parameter_Position - Position
            torque = mathsUtility.CrossProduct(radiusVector,
parameter_Force)
            AddTorque(torque);
        ENDMETHOD
        METHOD AddTorque
            angularVelocity += (parameter_Torque * InverseInertia)
        ENDMETHOD
        INSTANTIATE PhysicsBody
ENDCLASS
```

Data Dictionaries

Each object within the program will have a variety of attributes associated with it (mass, velocity, material, etc.). As well as there being some attributes that are associated with the environment itself (e.g. gravity). Below shows what is stored in each class. Note that all of these can be passed from one class to another, this simply shows where the original attribute is stored.

Program Manager

Identifier	Data Type	Purpose
SCREEN_WIDTH	Integer	Store the width of the screen
SCREEN_HEIGHT	Integer	Store the height of the screen
NR_OF_PARTICLES	Integer	Store number of shapes that exist initially (To be randomly generated)
texture_Particle	Texture2D	Store the texture for particles
my_Shape	Texture2D	Store the texture for the shapes
texture_Background	Texture2D	Store the texture for the background
my_DeltaTime	Float	Store time program has been running (used for calculations)
my_ObjectsManager	ObjectsManager	An instance of the ObjectsManager class
my_PhysicsManager	PhysicsManager	An instance of the PhysicsManager class
my_InputManager	InputManager	An instance of the InputManager class
my_RenderManager	RenderManager	An instance of the RenderManager class
my_RandomGenerator	Random	Used for random generation of initial particles
my_Graphics	GraphicsDeviceManager	Sets up the graphics device manager
vertices	Vector2[]	Set of vertices used in loading content and presets
vertices2	Vector2[]	Set of vertices used in loading content and presets
vertices3	Vector2[]	Set of vertices used in loading content and presets

vertices4	Vector2[]	Set of vertices used in loading content and presets
lastUpdate	Double	Time of last update
lastDraw	Double	Time that last shape was drawn
drawingCircle	Boolean	Store whether circle drawing mode is active
drawingPolygon	Boolean	Store whether polygon drawing mode is active
showControls	Boolean	Store whether the control panel should be displayed
isStatic	Boolean	Stores whether the object is affected by forces or not
preset	Integer	Stores the current pre-set that is being displayed
circleRadius	Float	Stores current radius of circle being drawn
circleStartPosition	Vector2	Stores centre of circle being drawn
polygonVertices	List<Vector2>	List of co-ordinates corresponding to the points of the polygon being drawn
normal1	Vector2	The normal vector to a polygon edge, used to check whether next point can be placed in mouse's current position
normal2	Vector2	The normal vector to a polygon edge, used to check whether next point can be placed in mouse's current position
normal3	Vector2	The normal vector to a polygon edge, used to check whether next point can be placed in mouse's current position
newEdge1	Vector2	An edge that would be formed if the point was placed at the current mouse position, used to

		check whether position is valid.
newEdge2	Vector2	An edge that would be formed if the point was placed at the current mouse position, used to check whether position is valid.
edge	Vector2	Edge that would be formed if point placed in current position, used to check if position is valid. (When only 2 points have been placed)
normal	Vector2	The normal vector to a polygon edge, used to check whether next point can be placed in mouse's current position (When only 2 points have been placed)

Render Manager

Identifier	**Data Type**	**Purpose**
my_SpriteBatch	SpriteBatch	Used for drawing objects
my_Strings	Queue<String,Vector2>	Queue of strings to be rendered
my_RenderObjects	Queue<RenderObject>	Queue of shapes to be rendered
my_SpriteFont	SpriteFont	Store texture for strings
my_PixelWhite	Texture2D	Store texture for polygons
my_CircleWhite	Texture2D	Store texture for circles
j	Integer	Used in a loop to assign string placement
line	RenderLine	Reference to the RenderLine class been rendered
lineVector	Vector2	Contains the vector between the two ends of the line
lineLength	Integer	Stores the length of

		the line
perpendicularDotProduct	Float	Stores the dot product of the unit vector and the line vector, this is used in drawing the line.

Input Manager

Identifier	Data Type	Purpose
my_PreviousKeyboardState	KeyboardState	Stores the state of the keyboard from the last update
my_PreviousMouseState	MouseState	Stores the state of the mouse from the last update

Objects Manager

Identifier	Data Type	Purpose
my_Objects	List<GameObject>	A list of the instances of GameObject

Game Object

Identifier	Data Type	Purpose
my_Body	PhysicsBody	Reference to the corresponding PhysicsBody
my_Texture	Texture2D	The texture of the object
my_Colour	Color	The colour of the object

Physics Manager

Identifier	Data Type	Purpose
minimum	Float	Minimum point of the object's AABB
maximum	Float	Maximum point of the object's AABB
collisionInfo	CollisionInfo	Information about the collision (contains other attributes)
CollisionNormal (*)	Vector2	The normal to the collision vector at the collision point
Overlapping (*)	Vector2	
CollisionPoints (*)	List<Vector2>	List of colliding points

IntersectionPoints (*)	List<Vector2>	List of points which are overlapping
my_CollisionListeners	CollisionInfoDelegate	Used to store collisionInfo and bodies involved. Then delegates to certain functions
my_CollisionFunctionParameters	IsCollidingDelegate	Used to store collisionInfo (by reference) and bodies involved. Then delegates to certain functions
my_ResolveFunctionParameters	CollisionResolveDelegate	Used to store collisionInfo and bodies involved. Then delegates to resolve function
my_Bodies	List<PhysicsBody>	List of physics bodies
my_Gravity	Vector2	The gravity vector
my_GravityNormal	Vector2	The vector 90° to the gravity vector
bodySelected	PhysicsBody	Stores the body currently selected which will be inspected by the debug
lastMousePosition	Vector2	Stores the location of the mouse on the last update
my_InputManager	InputManager	An instance of the input manager
showDebug	Boolean	Stores whether the debug information should be displayed
mousePosition	Vector2	Stores the mouse's current position
CentreOfGravity	Vector2	Used to store the point where gravity should be applied to a polygon
distance	Vector2	Stores distance between centres of two circles
widths	Float	Stores the addition of the two circles' radii
specialDistance	Vector2	Used to determine a

		collision normal when two circles occupy the exact same position
overlappingAmount	Float	Stores how much the two circles are overlapping
verticesInside	List<Vector2>	Stores which vertices of the second polygon are inside the first polygon
edge	Vector2	Stores a vector representing a polygon edge
edgeNormal	Vector2	Stores the vector perpendicular to edge
shortestOverlapAxis	Vector2	Stores the normal of the edge of the 2nd polygon that overlaps the 1st polygon the least
projectionA	Projection	Stores the AABB's maximum and minimum points of the 1st polygon as well as whether it overlaps with the 2nd polygon
projectionB	Projection	Stores the AABB's maximum and minimum points of the 2nd polygon as well as whether it overlaps with the 1st polygon
shortestOverlapAmount	Float	The least amount overlapped by any edge of the 2nd polygon
overlapAmount	Float	Amount overlapped by the current edge of the 2nd polygon
point1	Vector2	Stores one point of current 2 edges
point2	Vector2	Stores one point of current 2 edges
point3	Vector2	Stores one point of

		current 2 edges
point4	Vector2	Stores one point of current 2 edges
vertex	Vector2	Stores vertex which is colliding
dot	Float	Stores dot product of vertex and edgeNormal
closestProjection	Vector2	Stores the closest vector of the closest polygon projection to the circle
insidePolygon	Boolean	Stores whether the circle is inside the polygon
vectorToCircle	Vector2	Stores the vector from the centre of the circle to the polygon vertex
currentEdge	Vector2	Stores the edge of the polygon currently being checked
currentEdgeLengthSquared	Float	The length of the current edge squared
circleToEdgeProjection	Float	The projection from the centre of the circle to the edge of the polygon being checked
vectorCircleProjectionEdge	Vector2	The vector corresponding to the current edge and the circle-to-edge projection
projectionToCircle	Vector2	Stores the projection to the circle from the point
projectionToCircleLengthSquared	Float	The square of the length of the projection to the circle
j	Float	Stores the impulse exerted on an object
jt	Float	Stores the frictional force exerted on an

		object
CoefficientOfFriction	Float	Stores the frictional coefficient of the collision
collisionPointRelativeVelocityAtoB	Vector2	The relative velocity between 2 colliding bodies
collisionTangent	Vector2	The tangent vector to the collision
collisionPointRadiusA	Vector2	The vector between the 1st body and the collision point
collisionPointRadiusB	Vector2	The vector between the 2nd body and the collision point
collisionPointRadiusAPerpendicular	Vector2	The vector perpendicular to collisionPointRadiusA
collisionPointRadiusBPerpendicular	Vector2	The vector perpendicular to collisionPointRadiusB
collisionPointVelocityA	Vector2	The velocity of the first body
collisionPointVelocityB	Vector2	The velocity of the second body

(*) Contained within the CollisionInfo structure

Physics Body

Identifier	**Data Type**	**Purpose**
my_ObjectCount	Integer	Keep track of the number of physics objects currently active in the application
ShapeDefine	PhysicsShape	Holds data about the body that aren't specific to physics or a particular shape type.
CircleDefine	PhysicsCircleDefine	Holds data about the body that is particular to it being a circle.
PolygonDefine	PhysicsPolygonDefine	Holds data about the body that is particular to it being a polygon.
my_ID	Integer	A unique identifier for the physics object
my_Force	Vector2	The vector of the

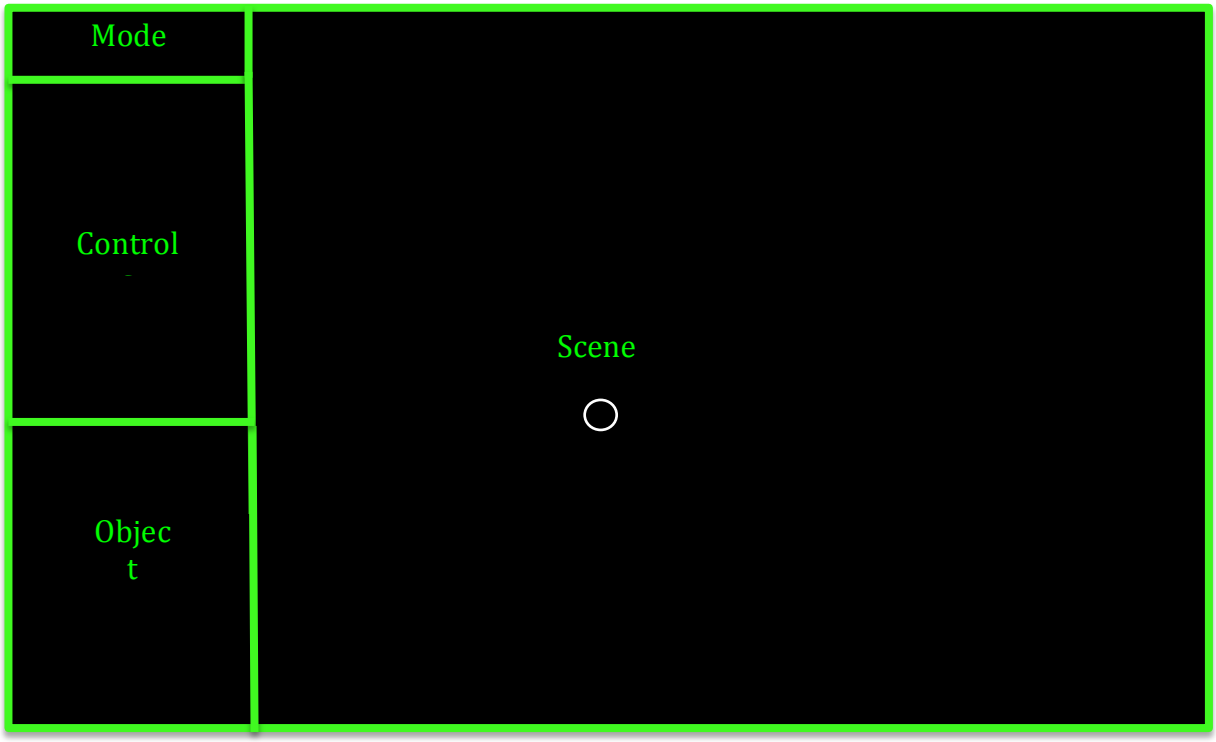
		current force acting on the physics object
my_Acceleration	Vector2	The vector of the current acceleration of the physics object
my_LinearVelocity	Vector2	The vector of the current velocity of the physics object
my_AngularVelocity	Float	A float that stores the current angular velocity of an object.
my_Position	Vector2	The co-ordinates of the physics object's current position
my_LastPosition	Vector2	The co-ordinates of the physics object's position during the last update
my_InverseMass	Float	1 / The Mass of the Physics Object. Used for calculations.
my_InverseInertia	Float	1 / The Inertia of the Physics Object. Used for rotation calculations.

UI Diagrams

Whilst the project is heavily centred on the physics being calculated by the engine, I do still require an interface on which to display and test it. A design of the interface is given below.

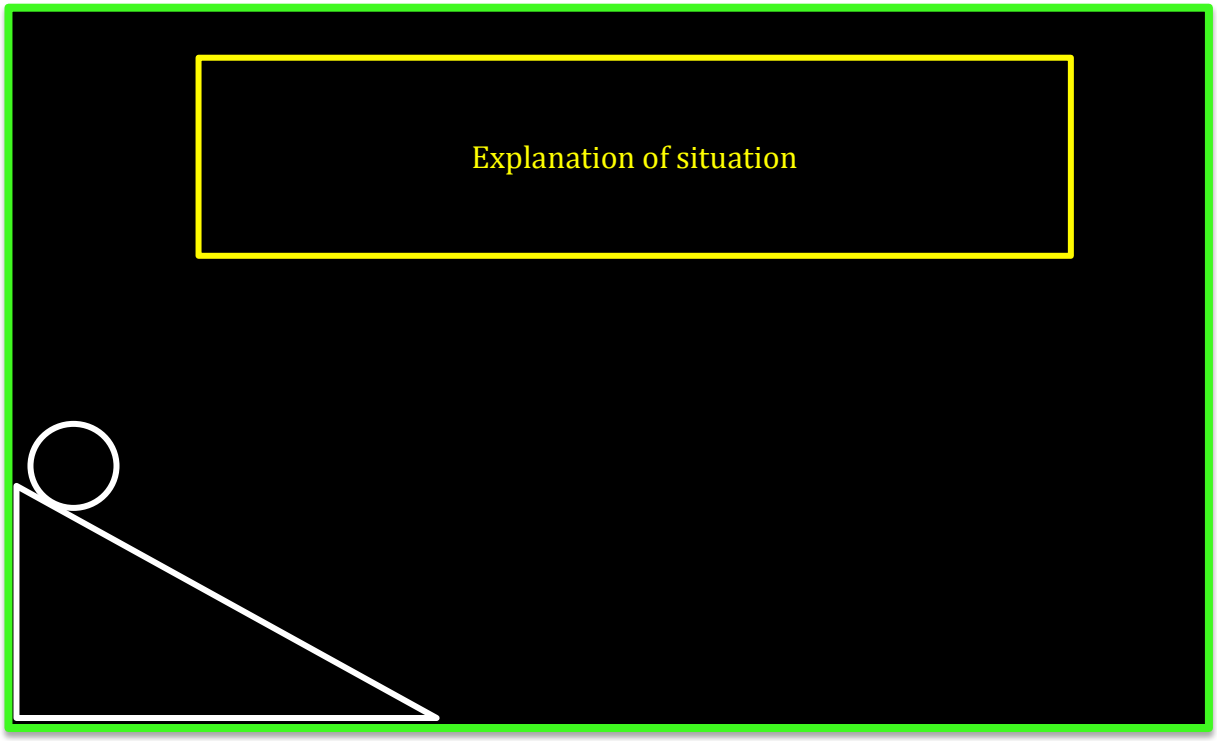
Sandbox Layout

The white circle represents the cursor, when in drawing mode the circle will be green when a point can be placed at the current location and red when it cannot.

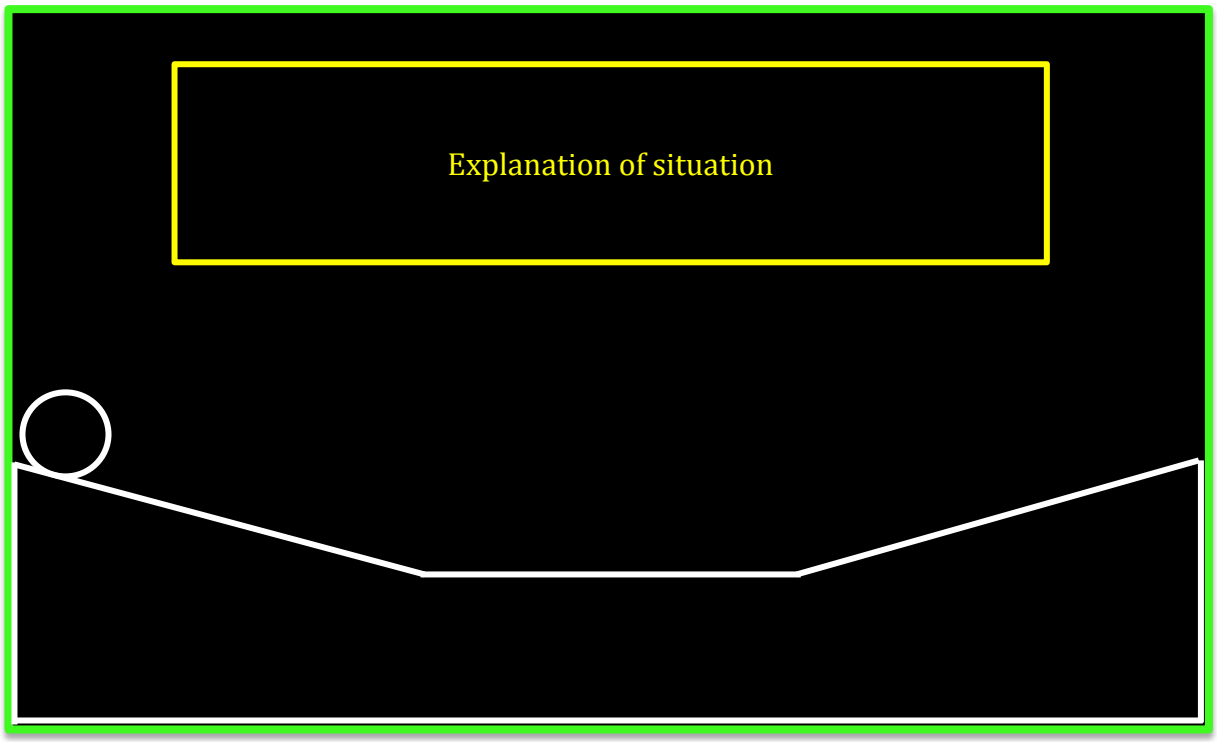


Pre-set 1

The first pre-set scenario will demonstrate a ball rolling down a ramp and it should fall short of the edge of the scene due to frictional forces. Text explaining the physics of what's going on will be given above it.

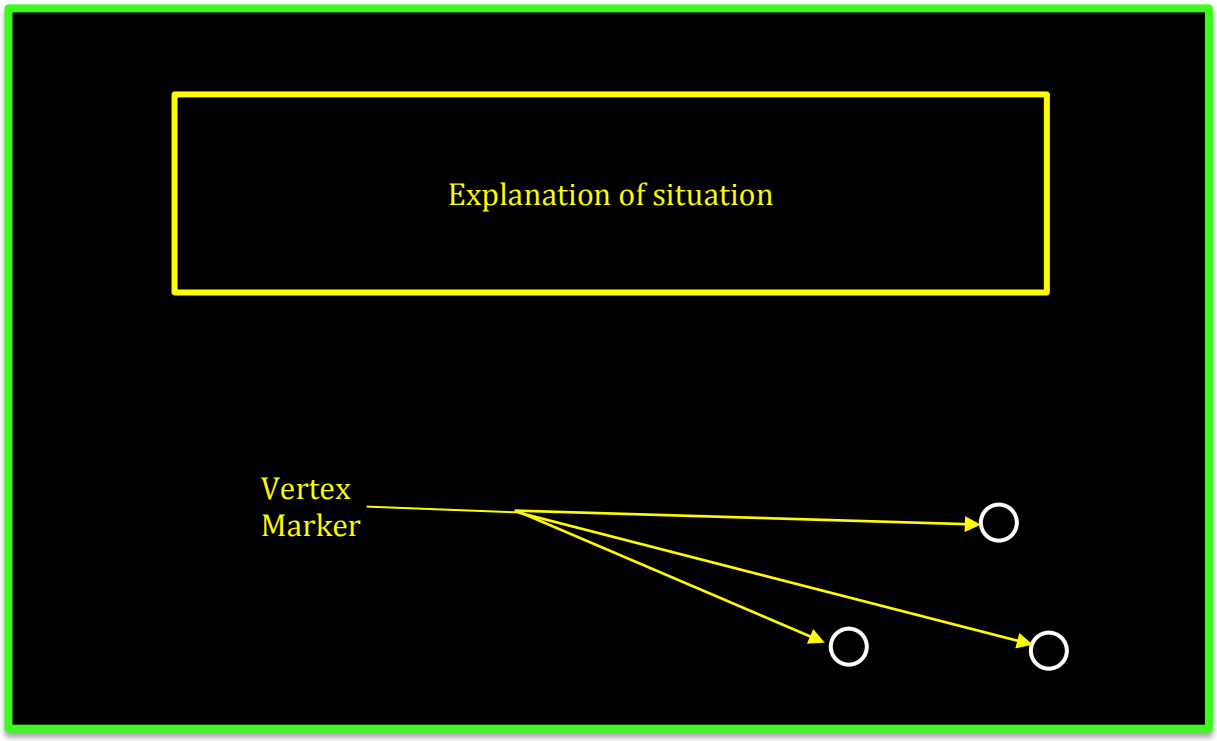


Pre-set 2



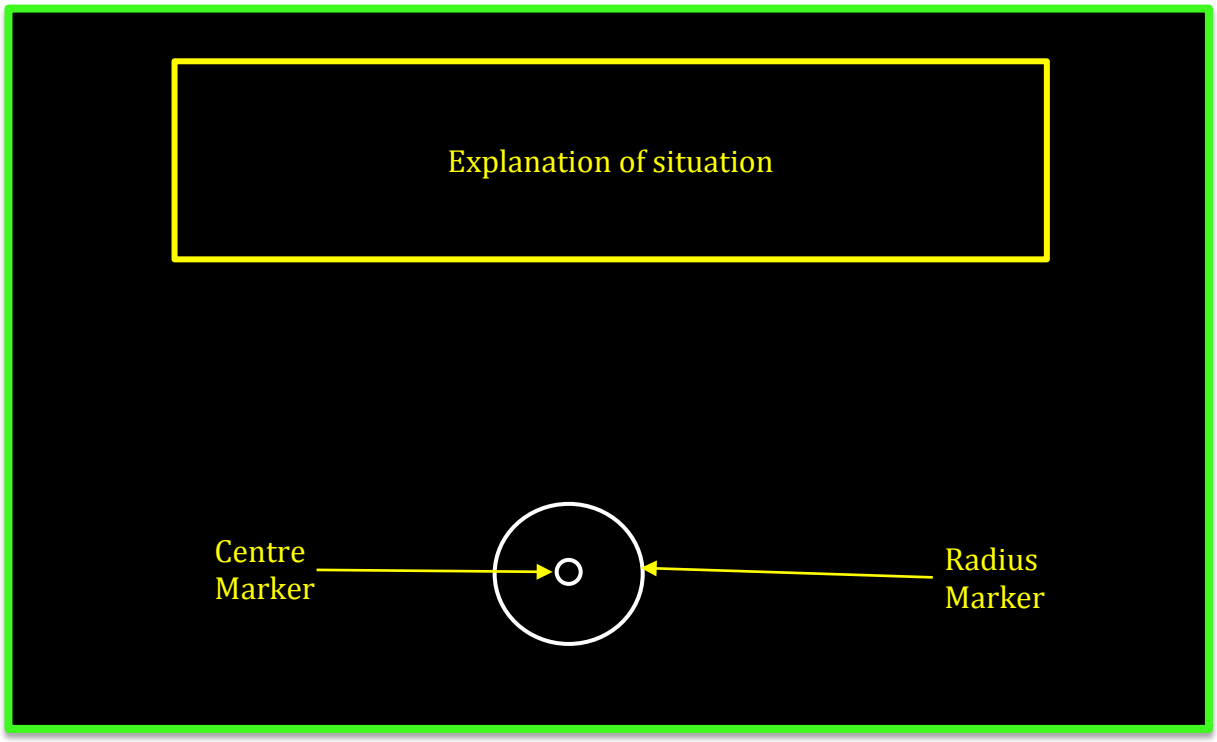
The second pre-set scenario will show a ball rolling down one end of the structure and attempting to roll up the other side, it should begin to roll up but not make it to the top – due to a lack of kinetic energy (which shouldn't be enough to do work against all the resistances) – and then roll back down. It should eventually come to a rest in the middle after a few oscillations about the middle. An explanation of the situation will be given above.

Pre-set 3



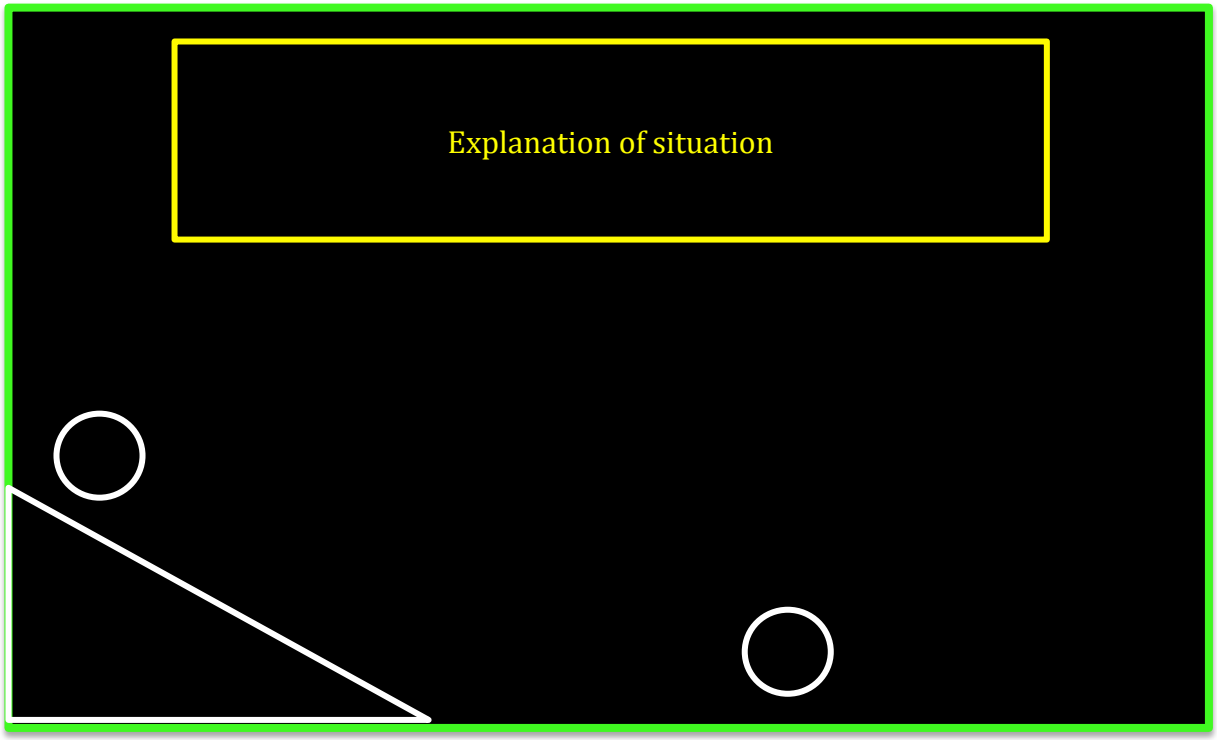
The third pre-set scenario will give a quick tutorial on how to use the polygon drawing tool by giving three markers and an explanatory text on how to use the tool. The user will then use the markers to draw a polygon.

Pre-set 4



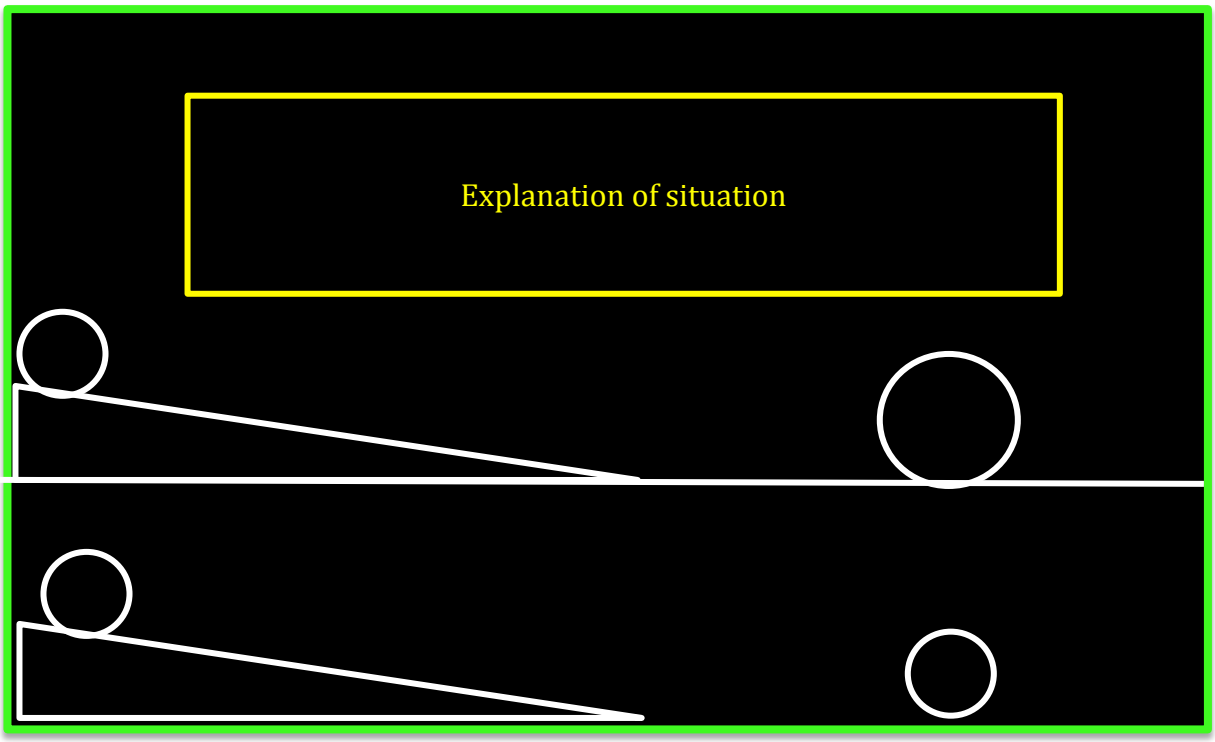
The fourth pre-set scenario is another drawing tutorial but this time it will teach the user about the circle drawing tool, a centre marker is given for a point to start from and a radius marker is given for aiding in setting the radius. An explanation of how to do the tutorial will be given.

Pre-set 5



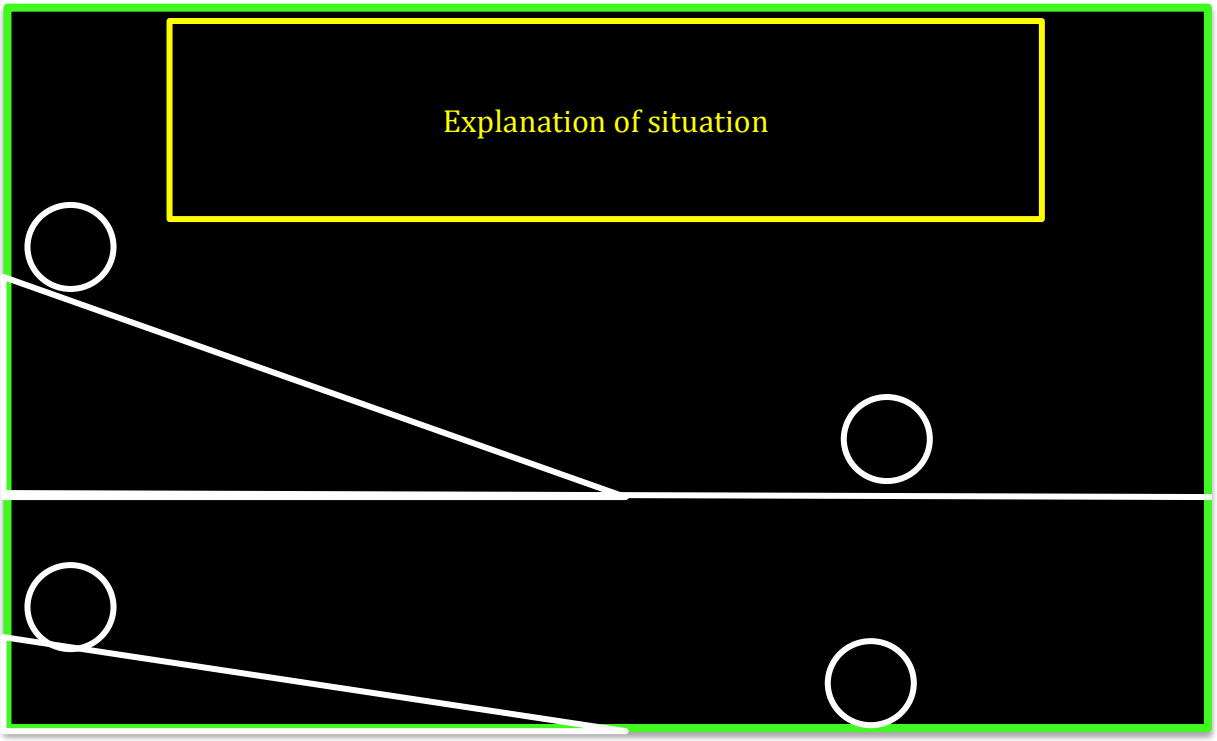
The fifth scenario will show two balls of equal size, one rolling down a ramp and along the floor into the other. The collision should cause the 2nd ball to move and cause the 1st one to slow down. An explanation of the scenario will be given.

Pre-set 6



The sixth scenario will involve two balls rolling down ramps of equal gradient, one will roll into a ball of equal size and another will roll into a ball that is twice the radius but is an equal distance away. The larger ball should move less distance due to its greater mass. An explanation of the scenario will be given.

Pre-set 7



The seventh scenario will involve two balls rolling down ramps of different gradient, both will roll into a ball of equal size an equal distance away. The top ball should move further as it has been hit with a greater speed. An explanation of the scenario will be given.

Language Choice and Libraries

Whilst this type of project can be undertaken in a wide selection of languages, I've chosen to utilise C#. This is because it's purely object-oriented allowing me to focus on using the one paradigm. Along with this, it's type safe meaning it deals well with memory leaks so not as much focus is needed on preventing problems due to memory locations being accessed by invalid objects.
It also has a wealth of libraries at its disposal meaning I can carry out certain operations with ease that might be harder to do in other languages. On top of this, I have quite a bit of experience in C# so I won't need to learn a new language and this will save time.
The libraries I will make use of are the system libraries and the XNA game studio libraries. The system libraries contain a large number of operations such as Vector2 operations (found in System.Numerics), which I will be relying on extensively for my calculations. The XNA game studio libraries are centred around making game development easier, this means it contains functions which help with rendering and inputs which will help shorten time spent on less important elements of the project.

Prototyping Plan

It's not feasible to make the whole program at once and then test it at the end so I've decided to split the program into a series of prototypes, which culminate in the final product. The reason for making prototypes is so I can fix errors as I build.

Version	Features
0.1	Gravity, Shape Drawing
0.2	V0.1 + Collision Detection and Collision Resolution
0.3	V0.2 + Rotation
0.4	V0.3 + Friction
0.5	V0.4 + Any extra features (Possible ideas include manipulation of gravity, Newton's Law of gravitation and a material table)

Testing Plan

In order to test my system I will carry out a series of tests that check whether gravity works as intended, that I can draw shapes only in valid locations, that collision is detected and resolved as planned, that the shapes rotate as planned and that the friction works as intended. I will record the screen as I carry out these tests and use that as evidence – as well as providing screenshots and any other relevant details.
I will then allow other users to test out the software and look for any issues that I may have missed. I will take any feedback given by them and act upon it.

Test	Expected Result
Starting the program.	Sandbox mode scene should be loaded.
Press P when polygon drawing mode isn't engaged.	String indicating that polygon drawing mode should appear as well as a cursor

	change.
Press P when polygon drawing mode is engaged when no points placed.	Polygon drawing mode should disengage.
In polygon drawing mode, move cursor into a valid position.	Cursor should be green.
In polygon drawing mode, move cursor into an invalid position.	Cursor should be red.
In polygon drawing mode, click when cursor is green.	Point should be placed, indicated by a circle.
In polygon drawing mode, click when cursor is red.	Point shouldn't be placed.
In polygon drawing mode, press P when 2 or less points have been placed.	Polygon mode should disengage and no polygon should be drawn.
In polygon drawing mode, press P when 3 or more points have been placed.	Polygon mode should disengage and a polygon should be created.
Press O when circle drawing mode isn't engaged.	String indicating that circle drawing mode should appear.
Press O when circle drawing mode is engaged and no circle has been drawn.	Circle drawing mode should disengage.
Click and drag when in circle drawing mode.	Should set radius of circle with centre at the clicked point, indicated by a green circle.
Press O when circle drawing mode is engaged and a circle has been drawn.	Circle drawing mode should disengage and a circle should be created.
Press ESC.	Program should close.
Press X when control help is on.	Control help should turn off (controls shouldn't be displayed)
Press X when control help is off.	Control help should turn on (controls should be displayed)
Press Q when isStatic is false.	isStatic should be true indicated by a string in the drawing modes.
Press Q when isStatic is true.	isStatic should be false indicated by a string in the drawing modes.
Draw a polygon when isStatic is true.	Polygon should stay where it is and be unaffected by gravity or impulses from collision.
Draw a polygon when isStatic is false.	Polygon should be affected by gravity and impulses from collision.
Draw a circle when isStatic is true.	Circle should stay where it is and be unaffected by gravity or impulses from collision.
Draw a circle when isStatic is false.	Circle should be affected by gravity and impulses from collision.
When debug is off, press M and click on the centre circle of an object.	Details about the object (debug) should be displayed.
Press M when debug is on.	Debug should turn off.

Let a polygon collide with the floor.	The polygon should be stopped, depending on the height dropped from it should bounce.
Let circle collide with the floor.	The circle should be stopped, depending on the height dropped from it should bounce.
Press 1	The 1st scenario should be loaded. The control panel and debug should turn off. The ball should roll down the ramp and fall well short of the wall. An explanation should be displayed.
Press 2	The 2nd scenario should be loaded. The control panel and debug should turn off. The ball should roll down one side and begin rolling up the other, it should then roll back down from a lower point. This should repeat until the ball comes to rest in the middle. An explanation should be displayed.
Press 3	The 3rd scenario should be loaded. The control panel and debug should turn off. An explanation should be displayed.
Press 4	The 4th scenario should be loaded. The control panel and debug should turn off. An explanation should be displayed.
Press 5	The 5th scenario should be loaded. The control panel and debug should turn off. The ball should roll down the ramp into the other ball causing the second ball to roll. An explanation should be displayed.
Press 6	The 6th scenario should be loaded. The control panel and debug should turn off. Two collisions should occur. The larger ball should stop closer to its start position than the smaller ball. An explanation should be displayed.
Press 7	The 7th scenario should be loaded. The control panel and debug should turn off. Two collisions should occur. The ball on the top layer should roll further than the ball on the bottom layer. An explanation should be displayed.
Allow a non-static object to collide with a static object.	Non-static object should bounce off and static object should be unaffected.
Allow 2 non-static polygons to collide.	Polygons should bounce off each other and rotate.

Allow 2 non-static circles to collide.	Circles should bounce off each other.
Allow a non-static polygon to collide with a non-static circle.	Objects should bounce off one another and polygon should rotate.

System Security

Whilst the system has no online functionality, some security still needs to be implemented. The main things that need to be considered are thread safety and memory leaks.

I need to prevent simultaneous access to the classes by multiple threads. But each of the classes is public so I shouldn't lock them. The reasoning for this is that a public object can be locked by anyone and this can lead to deadlocks. So I need to lock an internal object (AKA a property). C# has a syncroot property so I can lock that to take control of the locking pattern.

Memory leaks occur when a program incorrectly manages memory allocations so that a memory location that is no longer in use isn't released. This can exhaust available system memory and cause a buffer overflow. This can be avoided by using local variables effectively, making sure object references are necessary and making sure objects that are removed from the application are disposed of properly.

Calculations and Concepts

There is a large amount of classical mechanics that this program needs to simulate, this mainly comprises of Newton's Laws of Motion and momentum calculations.

Newton's Laws of Motion and SUVAT Equations

There are 3 integral laws that Newton defined for classical mechanics:

Newton's 1st Law: If the vector sum of all forces acting on an object is zero then the velocity of the object is constant.

Newton's 2nd Law: The rate of change of momentum of a body is directly proportional to the force applied and is in the same direction as the force. I.e. *Force = Mass * Acceleration*

Newton's 3rd Law: If Object A exerts a force on Object B, then Object B exerts a force on Object A of equal magnitude and opposite direction.

These will need to be acting at the centre of my engine. Since acceleration, velocity and displacement will need to be calculated from a pre-calculated force I can rearrange the second law to give:

*Acceleration = Force * (1/Mass)*

I can then utilise my acceleration in the SUVAT equation:

*Velocity = InitialVelocity + (Acceleration * Time)*

I can then substitute this into:

*Displacement = 0.5 * (InitialVelocity + Velocity) * Time*

But since this is updating rapidly I can sub in the previous velocity easily and the previous displacement using instantaneous velocity. All of that simplifies to give:

*Velocity += Acceleration * Time*
*Displacement += Velocity * Time*

This gives me a way of calculating the linear motion of the objects affected by the engine.

Vectors

In order to simulate these laws and equations in 2 dimensions, multiple values need to be stored as vectors in 2 dimensions. The two dimensions will represent the x (horizontal) and y (vertical) components of different mechanical properties. The following will be stored as vectors:

- Force
- Linear Velocity
- Acceleration
- Position

It's also important to understand the different operations that will be used with these vectors. The operations that will be used are:

- Addition/Subtraction
- Scalar Multiplication/Division
- Dot Product
- Cross Product

Addition and subtraction is fairly simple as demonstrated below:
(a, b) + (c, d) = (a+c, b+d)
(a, b) – (c, d) = (a-c, b-d)
Multiplication and division by a scalar also acts similarly to regular multiplication.
*k * (a, b) = (ka, kb)*
(a, b) / k = (a/k, b/k)
The dot product is different from scalar operations as it gives a 1-dimensional value from two vectors.

(a, b).(c, d) = ac + bd

This can be used to find the angle between two vectors by using:

$\underline{\mathbf{A}}.\underline{\mathbf{B}} = |\underline{\mathbf{A}}|*|\underline{\mathbf{B}}| * Cos\theta$

Where $\underline{\mathbf{A}}$ and $\underline{\mathbf{B}}$ are vectors, $|\underline{\mathbf{A}}|$ is the length of $\underline{\mathbf{A}}$ and Θ represents the angle between the 2 vectors.

The cross product is an operation normally used in 3 dimensions to find a vector perpendicular to the two given vectors. In 2 dimensions, however, the interpretation isn't as clear. The operation I'll be using for the 2 dimensional cross product gives a scalar and is given by:

*(a, b) x (c, d) = (a*d) – (c*b)*

Exactly what that value represents is unclear but it is used when calculating torque.

Calculating Restitution

Every collision has a coefficient of restitution (essentially a bounciness constant), this is calculated using:

$e = v/u$

Where v is the speed of separation of the two objects, u is the speed of approach of the two objects and e is the coefficient of restitution. The problem with this is it can only be calculated after a collision has occurred but we want to use it to calculate what happens during a collision.
There are 2 ways to go about this. Option 1 is to have a collision table which would store many different types of collision and pick the closest one every time a collision occurs. But this creates inconsistent physics and requires a large number of different collision variations to be hard-coded into the program.
Option 2 is to give each object a "bounciness scalar", this can be calculated as objects are created. Then on collision, we can get the scalar of each object involved and use a formula to deduce a coefficient of restitution.
There are many formulae I could use:

- e = min(Scalar1, Scalar2)
- e = max(Scalar1, Scalar2)
- e = (Scalar1 + Scalar2) / 2

To me, just taking a minimum or a maximum of the scalars is kind of simplistic and this results in the restitution being absurd in certain cases. I believe finding a mean will give a more accurate restitution. So I will use *e = (Scalar1 + Scalar2)/2*.

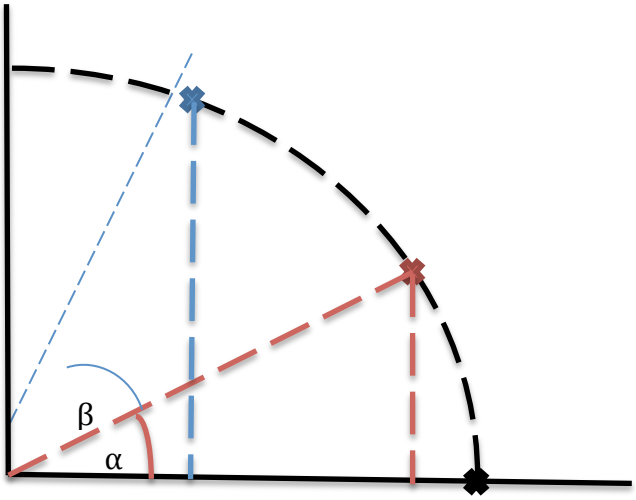
Rotation

In order to create a realistic environment, the objects will need to be able to rotate. First of all, we need to consider the properties of an object linked to rotation. The basic properties include rotational forces such as torque, angular speed and angular acceleration. So, we need to know how to calculate these properties. Torque can be calculated using:
*Torque = Radius * AngularVelocity*
Change In Torque = Force x Radius
Now, whilst this works with the same radius for all points on a circle, the same cannot be said for all points on a polygon's perimeter. Thus every single unique point on an object will need to be represented by a different 'radius'.
Linear velocity is the velocity of the centre of mass, not all of the points in an object. When rotation is introduced the points farther from the centre of mass rotate faster than those closer to the centre of mass. So in order to find the angular velocity of a specific point we use the equation:
AngularVelocity = LinearVelocity / Radius
And we can rearrange this to find the linear velocity we require:
*LinearVelocity = AngularVelocity * Radius*
As well as this, we need to be able to manipulate the points of an object so that it rotates. Using an Oriented Bounding Box (OBB) I can represent the height and

width of the object as vectors and these can be rotated using a 2x2 rotation matrix. A series of calculations will need to be done in order to get an angle which I can then put into a matrix of the form:

$$\begin{pmatrix} Cos\alpha & -Sin\alpha \\ Sin\alpha & Cos\alpha \end{pmatrix}$$ *Where α is an arbitrary angle*

The derivation for where this matrix comes from is given below.



= Point A (x_0,

= Point B (x_1,

= Point C (x_2,

Points A, B and C are equidistant from the origin and as can be seen from Point A that means they are all a distance x_0 from the origin. Hence:
By trigonometry in orange triangle:

$Cos\alpha = x_1 / x_0 , x_1 = x_0 * Cos\alpha$ (1)

$Sin\alpha = y_1 / x_0 , y_1 = x_0 * Sin\alpha$ (2)

By trigonometry in blue triangle

$Cos(\alpha+\beta) = x_2 / x_0 , x_2 = x_0 * Cos(\alpha+\beta) = (x_0 * Cos\alpha * Cos\beta) - (x_0 * Sin\alpha * Sin\beta)$
$= (x_1 * Cos\beta) - (y_1 * Sin\beta)$ *By (1) and (2)*

$Sin(\alpha+\beta) = y_2 / x_0 , y_2 = x_0 * Sin(\alpha+\beta) = (x_0 * Sin\alpha * Cos\beta) + (x_0 * Cos\alpha * Sin\beta)$
$= (y_1 * Cos\beta) + (x_1 * Sin\beta)$ *By (1) and (2)*

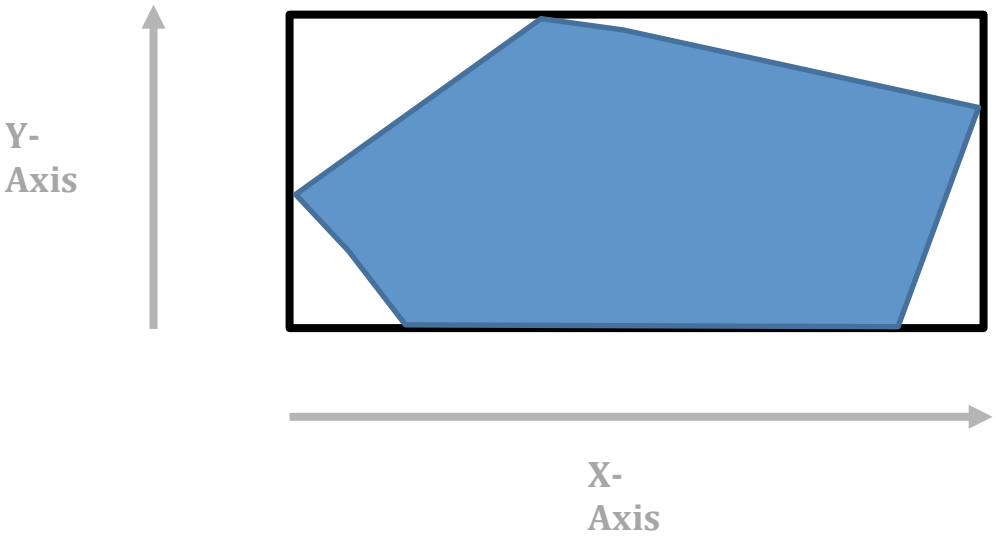
Thus $(x_2, y_2) = (x_1, y_1)\begin{pmatrix} Cos\beta & -Sin\beta \\ Sin\beta & Cos\beta \end{pmatrix}$

As a part of calculating the rotational forces I will need to calculate the inertia affecting an object. Again a circle is very simple and uses the formula:
$Inertia = 0.5 * Mass * Radius^2$
Once again polygons are more difficult to deal with and require a lot more work to calculate inertia accurately. It comes out to be something like this:

$Inertia = |(Mass * \sum (Vector_i * Vector_{i+1})) / (6 * \sum (Vector_i))|$

Collision Detection – Axis Aligned Bounding Boxes

An axis aligned bounding box is a box with 4 axes (or sides) which are lined up with the co-ordinate system (i.e. parallel to the x and y axes). This means that its rotation is locked and has perpendicular axes. It is often used as a set of boundaries for more complex shapes such as irregular polygons. Using a box which bounds more complex shapes is useful as it makes testing for intersections easier.

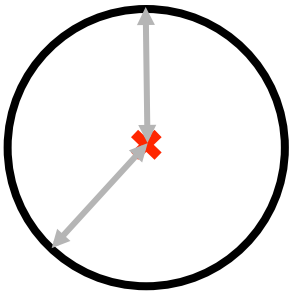


An Axis Aligned Bounding Box acting as a set of boundaries for an irregular pentagon.

Shapes can be set up in other ways to make collision more exact but this is often reserved for common shapes, in my case, I've only expressed a circle in a different form.

Collision Detection - Circles

Circles are a very common shape so are always given an exact collision. But they are much simpler to set up than many other shapes. A circle can simply be expressed as a central point position and a radius, this can then be used to calculate if it's intersecting with anything.



= Central Point
= Radius

Using this we can check if any point on the circumference is intersecting with another shape.

Finding Centre Points of Polygons

Whilst finding the centre of a circle is a very simple task for my project (there are built-in functions that do it) finding the centre of a polygon is a much more difficult task. The best way to do it is to treat it as an average of the points. Since I'm working with irregular polygons, the points of a polygon are stored as a list. Using these points a formula can be constructed that is to just find the mean.

$$Centre = (P_0 + P_1 ..... + P_k) / k$$

Where P_n is the vector of nth point in the list and k represents the number of vertices of the polygon, so the vector given will be:

$$((x_0 + x_1 ..... + x_k) / k , (y_0 + y_1 ..... + y_k) / k)$$

That is the current centre point of the polygon. This will need to be updated when the polygon's position or orientation is updated.

Mass of Shapes

Another thing to consider is the mass of each shape. Mass is related to volume by the following equation:

$$Mass = Volume * Density$$

Density is fairly simple to deal with as it is dependent on materials and a table of materials with certain properties can be kept, this can include density.
Calculating volume again is different when working with circles and polygons. In both cases, uniform thickness is presumed so instead of working with volume we're working with area.
To calculate the area of a circle we use the equation:

$$Area = \pi * radius^2$$

Once again, polygons are more complicated. The way it's going to be dealt with is by taking advantage of the fact that a polygon is stored as a list of points. Instead of calculating an exact area, an approximation can be calculated by finding the square difference between two adjacent points and adding them together.

A diagram showing how the area is approximated for an irregular polygon.

There is a problem with this method depending on the shape. Let's say that someone manages to draw a perfect square the area between each point would be 0 and as such the area would be 0. This would result in a floating object. In order to counteract this there are going to be constraints placed upon the physics of the engine, including a maximum/minimum area constraint.

Calculating Friction

Friction is one of the forces that needs to be applied when resolving collisions. Friction is a contact force which always opposes motion i.e. it acts in the opposite direction to the velocity. In real life, friction is a highly complex force and in order to model it I'm going to have to make large assumptions about how it works.

I'm going to model friction as an impulse, it will cause a velocity change in the direction of the negative tangent vector of the collision. The friction is fairly simple to calculate:

Friction = (-(1 + CoefficientOfRestitution)((VelocityB – VelocityA).TangentVector))/((1/MassA) + (1/MassB))

But in order to calculate this, we need to calculate the tangent vector which can be calculated using the normal vector. The following series of equations will work:

$Velocity^{Relative} = Velocity^{B} – Velocity^{A}$

$Tangent = Velocity^{Relative} – (Velocity^{Relative}. Normal) * Normal$

If I was to just use the equations I've given then friction would always be enough to stop movement, this isn't true and is stated by Coulomb's Law:

$Friction_{Max} = \mu * ContactForce_{Normal}$

Where μ is the coefficient of friction between the 2 objects. As such, the friction can never exceed this value but can be any value up to it.

Once again, in order to come up with a coefficient of friction, I will give everything a friction scalar and use a formula to determine a coefficient of friction for the collision.

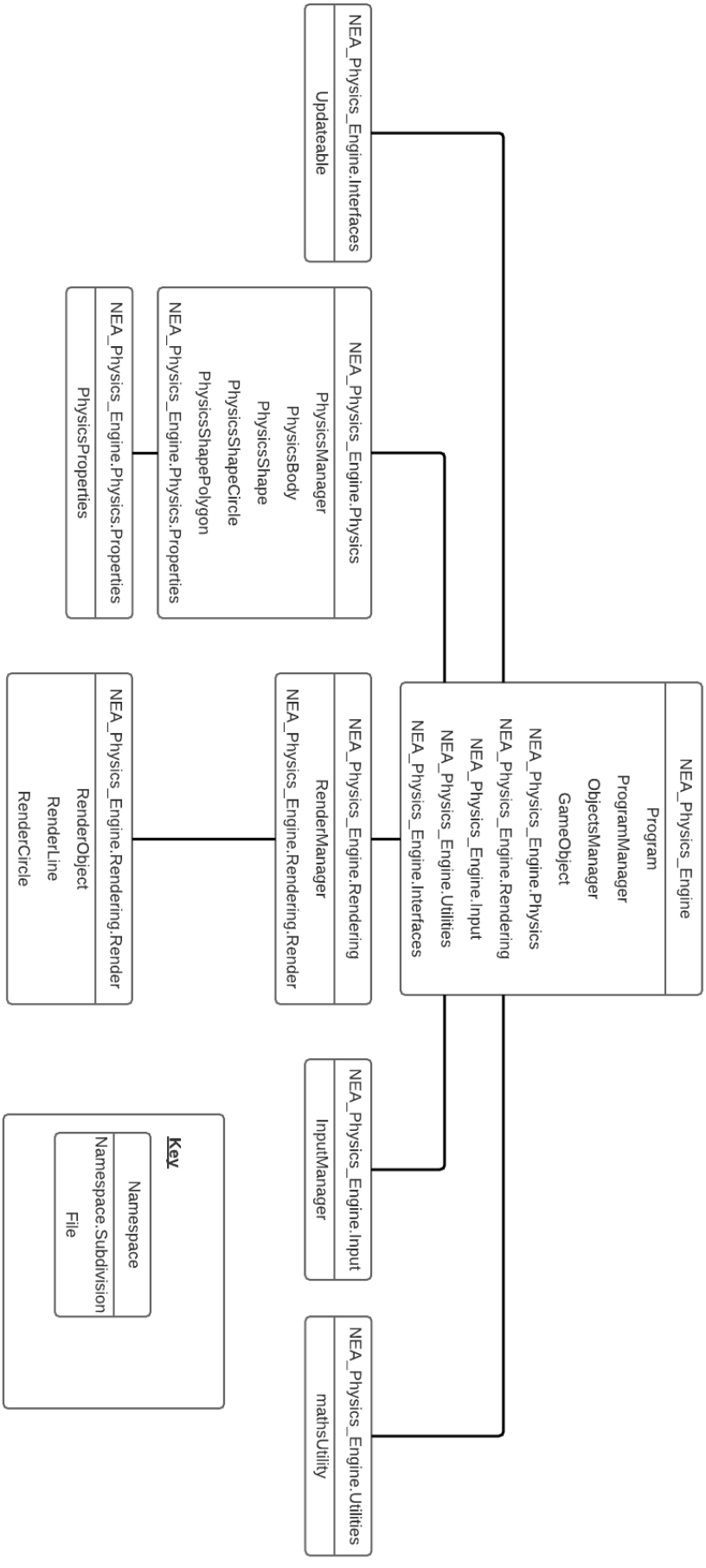
This doesn't cover everything though. Friction has what can be thought of as an activation energy and so objects will need a different coefficient of friction based on whether it's static or dynamic. We can then determine which to use for an interaction by the following code:

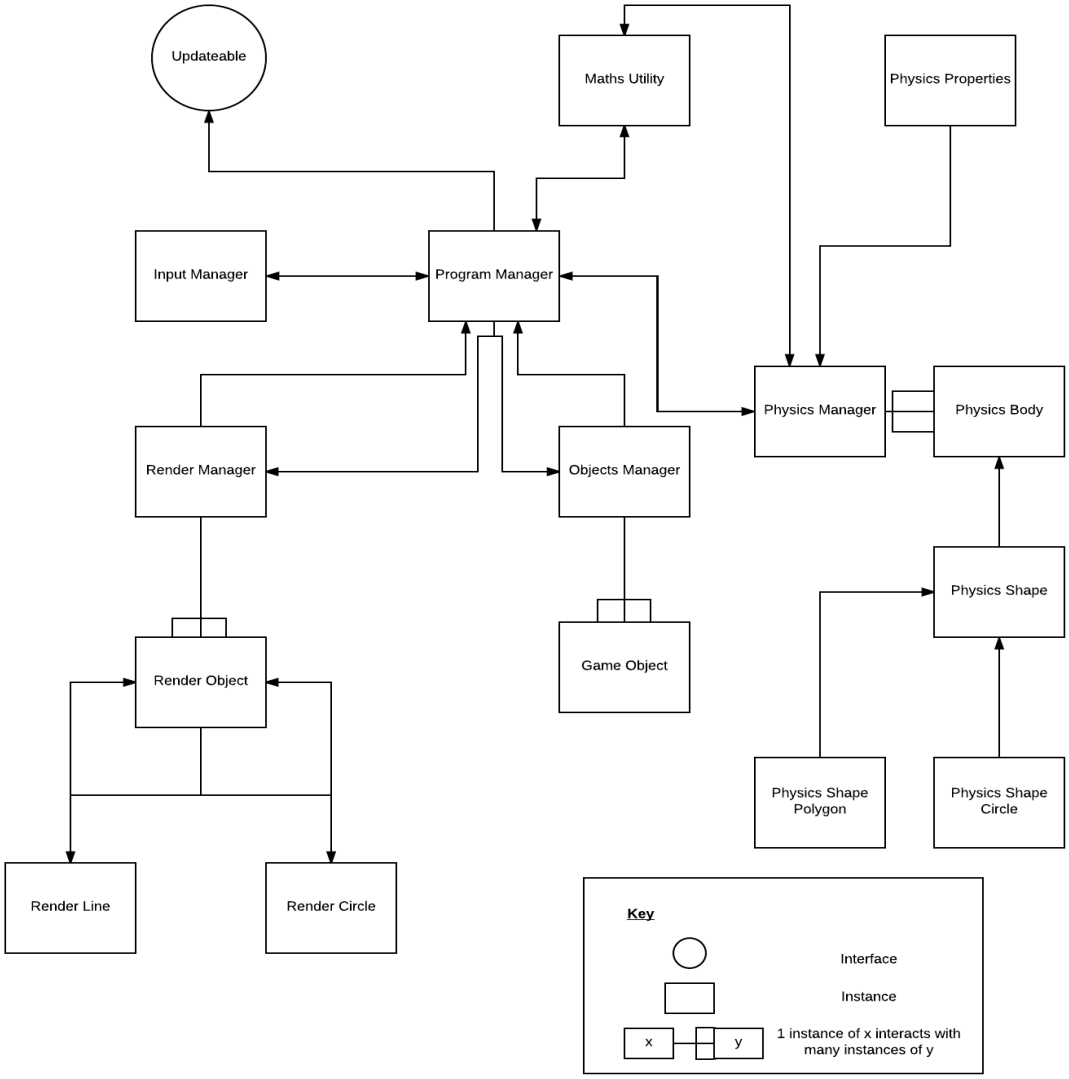
```
CoefficientOfFriction = (bodyA.StaticFriction + bodyB.StaticFriction) * 0.5f
IF (|jt| >= j * CoefficientOfFriction) THEN
                    CoefficientOfFriction = (bodyA.DynamicFriction +
bodyB.DynamicFriction) * 0.5f
                    jt = -j * CoefficientOfFriction
```

Where jt is the frictional impulse and j is the reaction impulse.

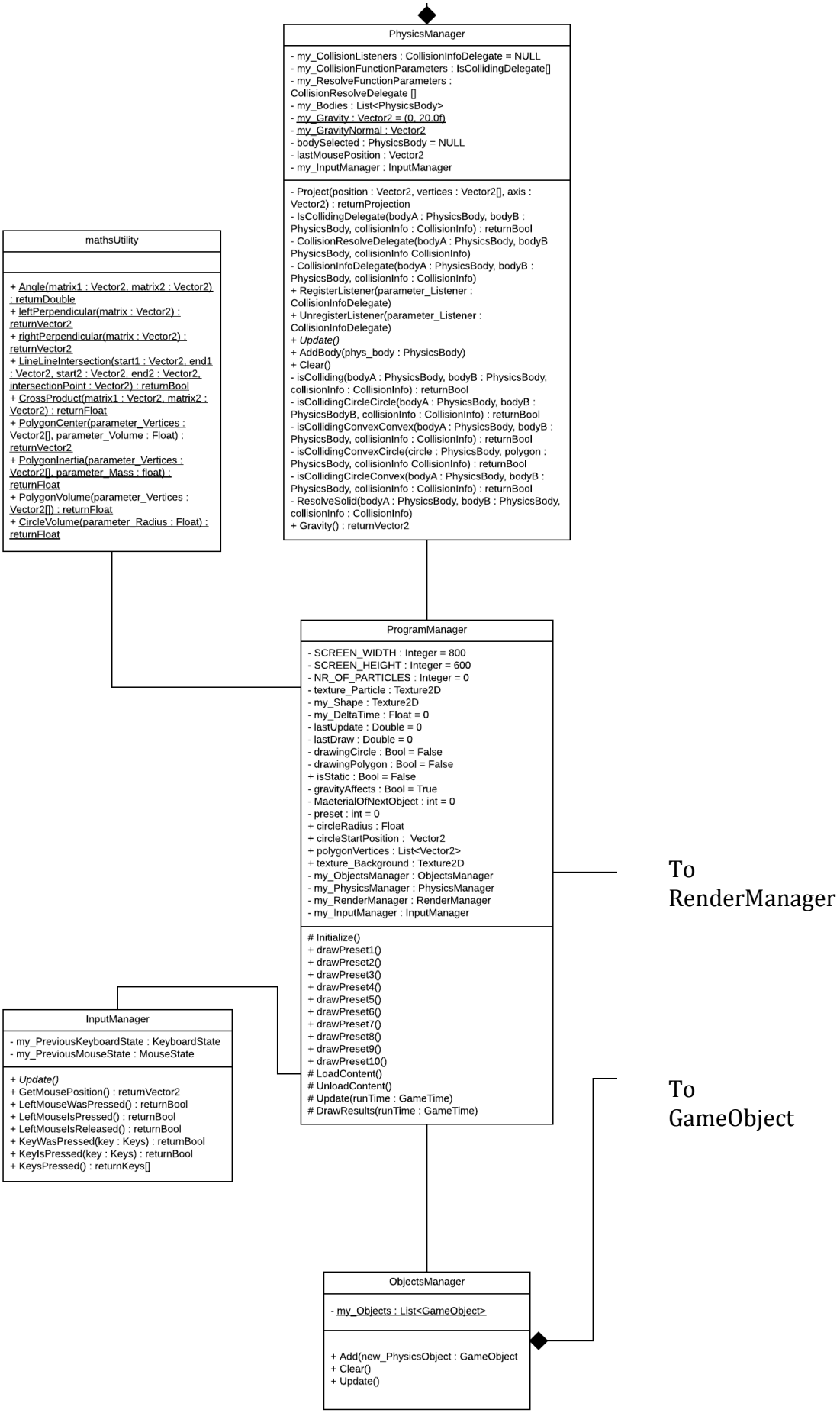
Implementation

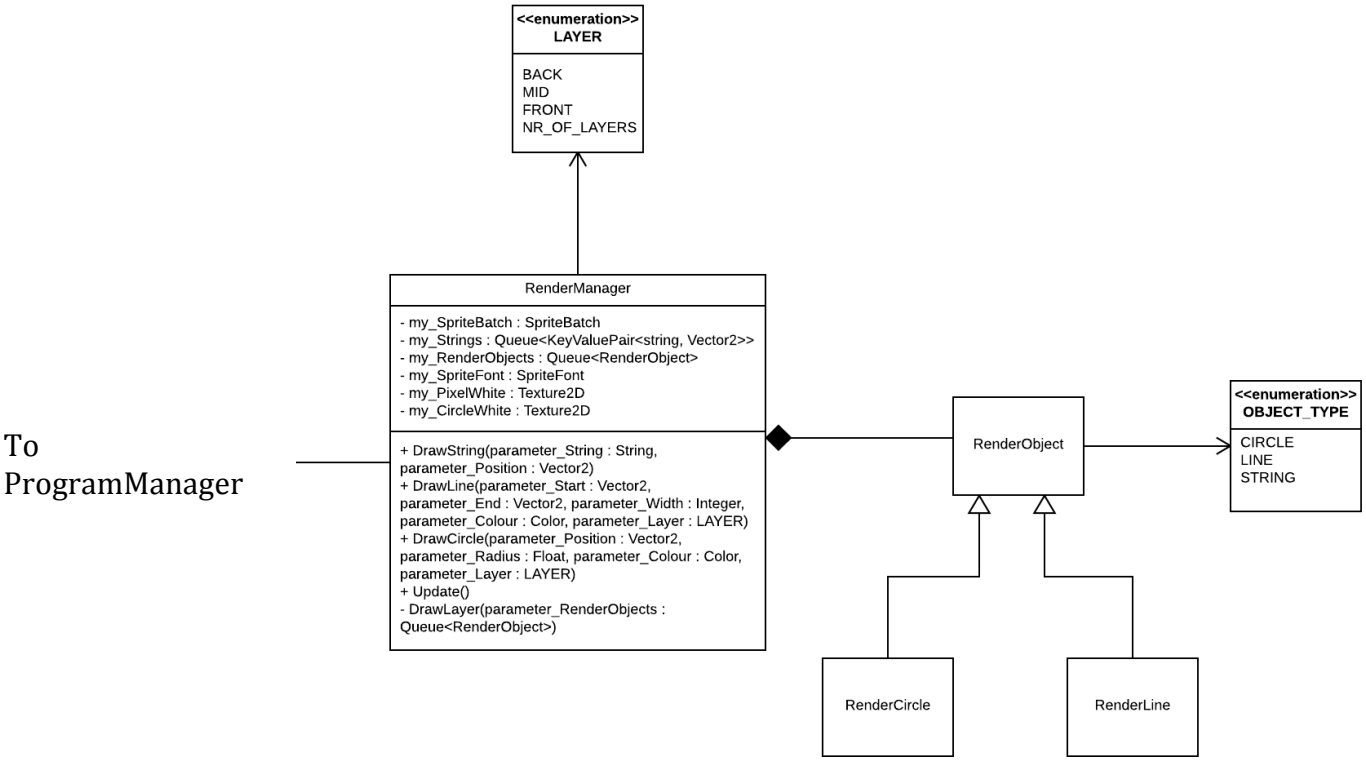
Updated Models

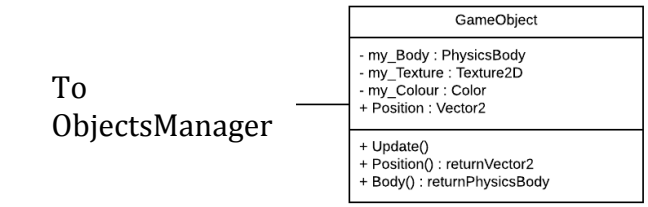


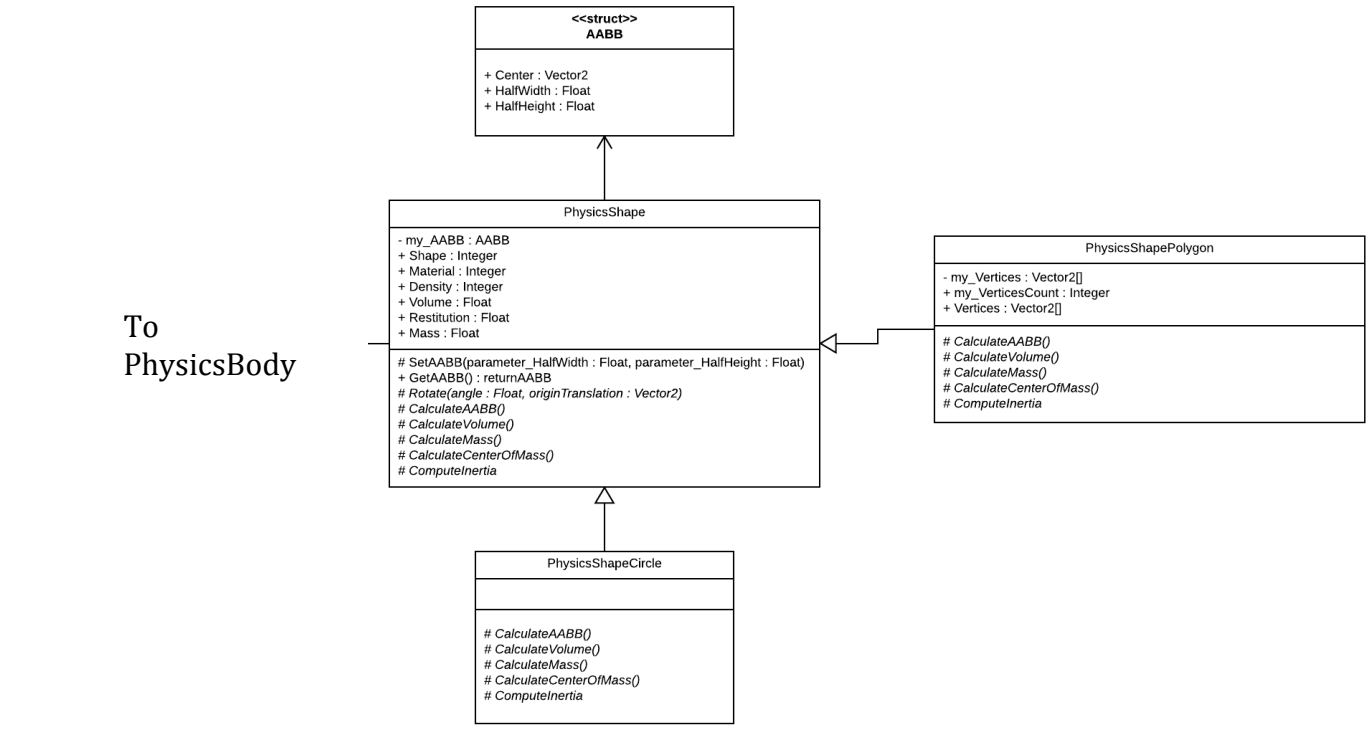

Updateable
Maths Utility
Physics Properties
Input Manager
Program Manager
Physics Manager
Physics Body
Render Manager
Objects Manager
Physics Shape
Render Object
Game Object
Physics Shape Polygon
Physics Shape Circle
Render Line
Render Circle
Key
Interface
Instance
x
y
1 instance of x interacts with many instances of y

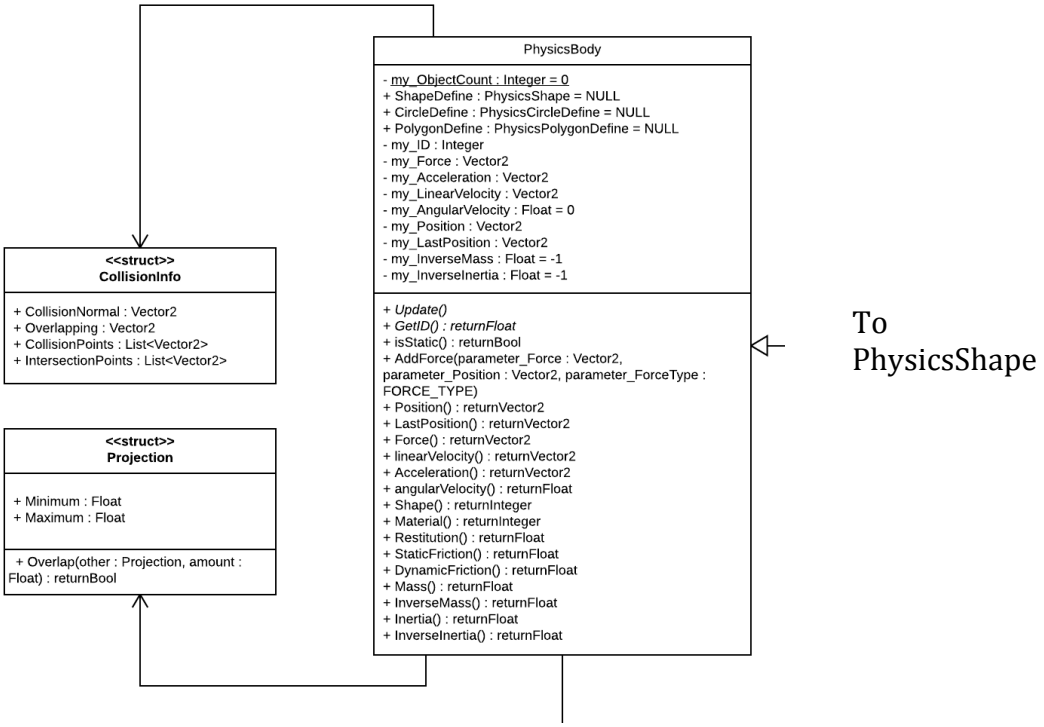
To PhysicsBody



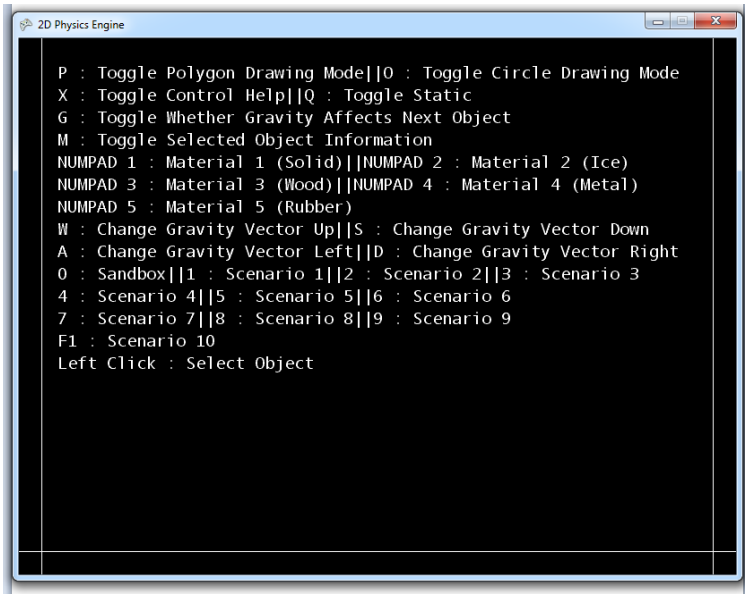

To ProgramManager
<<enumeration>>
LAYER
BACK
MID
FRONT
NR_OF_LAYERS
RenderManager
- my_SpriteBatch : SpriteBatch
- my_Strings : Queue<KeyValuePair<string, Vector2>>
- my_RenderObjects : Queue<RenderObject>
- my_SpriteFont : SpriteFont
- my_PixelWhite : Texture2D
- my_CircleWhite : Texture2D
+ DrawString(parameter_String : String, parameter_Position : Vector2)
+ DrawLine(parameter_Start : Vector2, parameter_End : Vector2, parameter_Width : Integer, parameter_Colour : Color, parameter_Layer : LAYER)
+ DrawCircle(parameter_Position : Vector2, parameter_Radius : Float, parameter_Colour : Color, parameter_Layer : LAYER)
+ Update()
- DrawLayer(parameter_RenderObjects : Queue<RenderObject>)
RenderObject
<<enumeration>>
OBJECT_TYPE
CIRCLE
LINE
STRING
RenderCircle
RenderLine

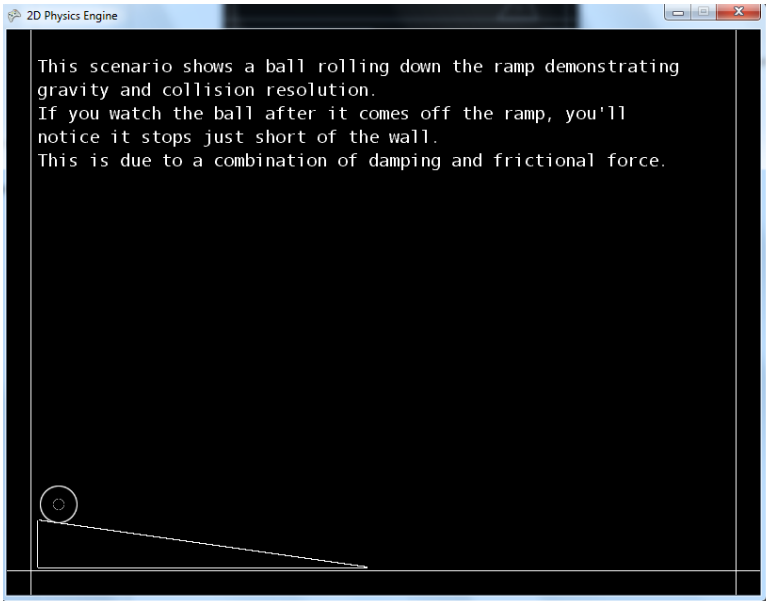

To ObjectsManager
GameObject
- my_Body : PhysicsBody
- my_Texture : Texture2D
- my_Colour : Color
+ Position : Vector2
+ Update()
+ Position() : returnVector2
+ Body() : returnPhysicsBody

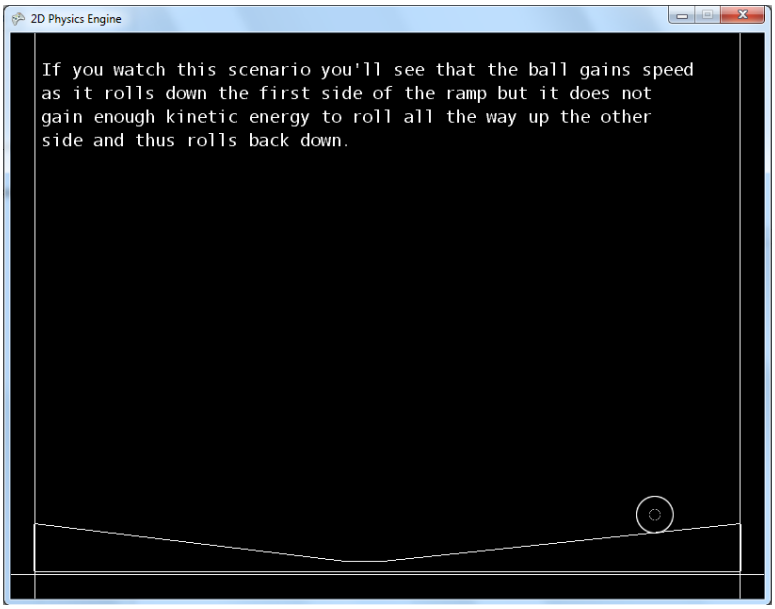

To PhysicsBody
<<struct>>
AABB
+ Center : Vector2
+ HalfWidth : Float
+ HalfHeight : Float
PhysicsShape
- my_AABB : AABB
+ Shape : Integer
+ Material : Integer
+ Density : Integer
+ Volume : Float
+ Restitution : Float
+ Mass : Float
# SetAABB(parameter_HalfWidth : Float, parameter_HalfHeight : Float)
+ GetAABB() : returnAABB
# Rotate(angle : Float, originTranslation : Vector2)
# CalculateAABB()
# CalculateVolume()
# CalculateMass()
# CalculateCenterOfMass()
# ComputeInertia
PhysicsShapePolygon
- my_Vertices : Vector2[]
+ my_VerticesCount : Integer
+ Vertices : Vector2[]
# CalculateAABB()
# CalculateVolume()
# CalculateMass()
# CalculateCenterOfMass()
# ComputeInertia
PhysicsShapeCircle
CalculateAABB()
CalculateVolume()
CalculateMass()
CalculateCenterOfMass()
ComputeInertia

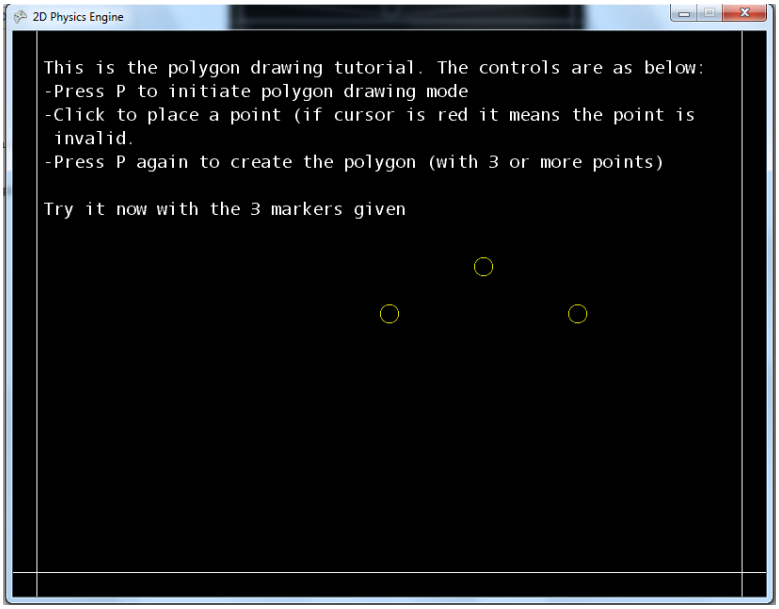

PhysicsBody
- my_ObjectCount : Integer = 0
+ ShapeDefine : PhysicsShape = NULL
+ CircleDefine : PhysicsCircleDefine = NULL
+ PolygonDefine : PhysicsPolygonDefine = NULL
- my_ID : Integer
- my_Force : Vector2
- my_Acceleration : Vector2
- my_LinearVelocity : Vector2
- my_AngularVelocity : Float = 0
- my_Position : Vector2
- my_LastPosition : Vector2
- my_InverseMass : Float = -1
- my_InverseInertia : Float = -1
+ Update()
+ GetID() : returnFloat
+ isStatic() : returnBool
+ AddForce(parameter_Force : Vector2, parameter_Position : Vector2, parameter_ForceType : FORCE_TYPE)
+ Position() : returnVector2
+ LastPosition() : returnVector2
+ Force() : returnVector2
+ linearVelocity() : returnVector2
+ Acceleration() : returnVector2
+ angularVelocity() : returnFloat
+ Shape() : returnInteger
+ Material() : returnInteger
+ Restitution() : returnFloat
+ StaticFriction() : returnFloat
+ DynamicFriction() : returnFloat
+ Mass() : returnFloat
+ InverseMass() : returnFloat
+ Inertia() : returnFloat
+ InverseInertia() : returnFloat
<<struct>>
CollisionInfo
+ CollisionNormal : Vector2
+ Overlapping : Vector2
+ CollisionPoints : List<Vector2>
+ IntersectionPoints : List<Vector2>
<<struct>>
Projection
+ Minimum : Float
+ Maximum : Float
+ Overlap(other : Projection, amount : Float) : returnBool
To PhysicsShape
To PhysicsManager

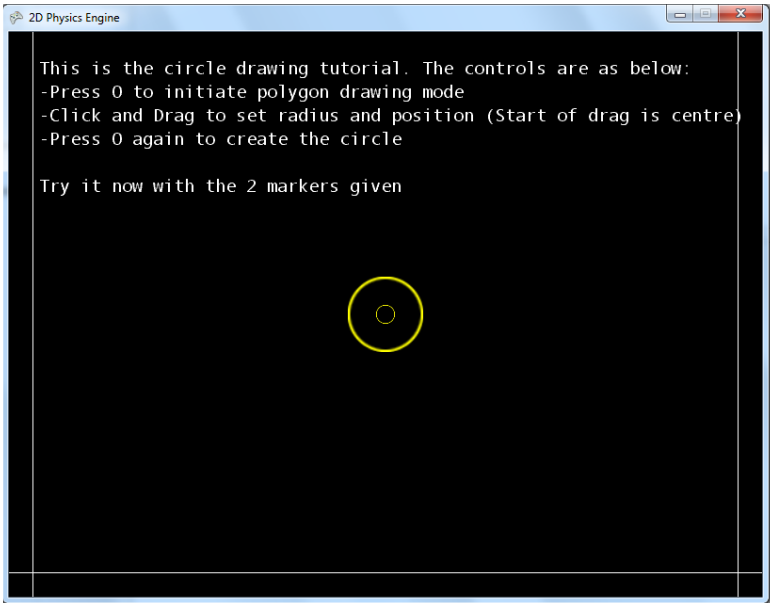
UI Screenshots

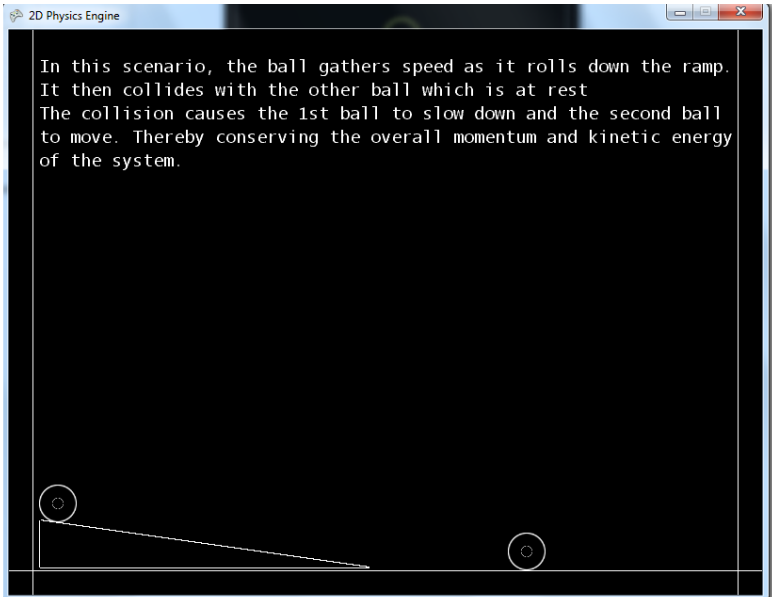


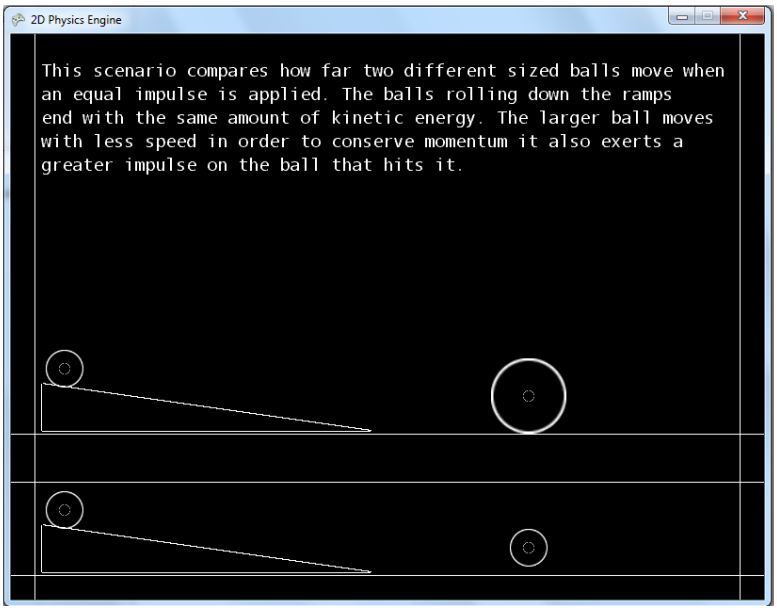


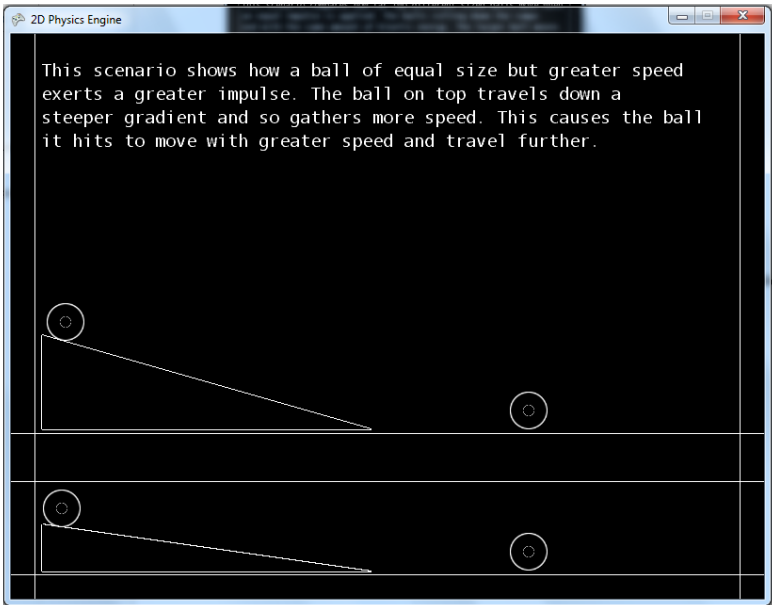


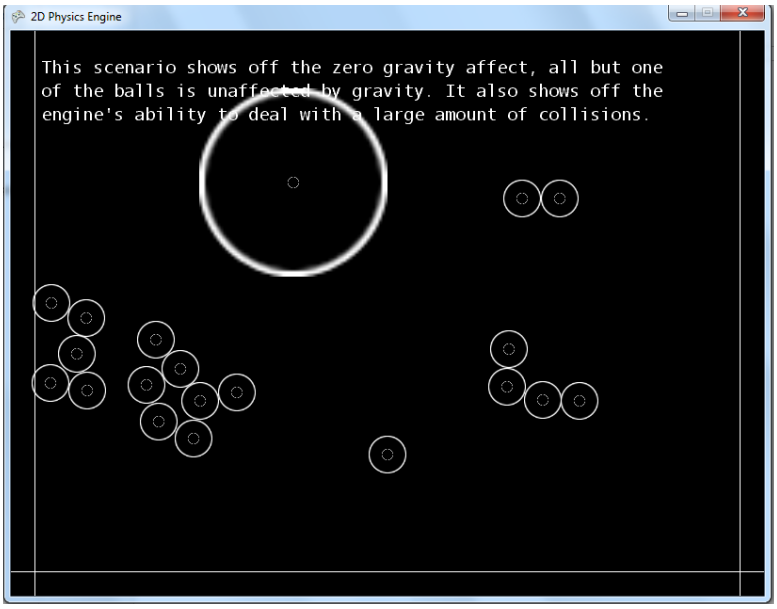

2D Physics Engine
This is the polygon drawing tutorial. The controls are as below:
-Press P to initiate polygon drawing mode
-Click to place a point (if cursor is red it means the point is
invalid.
-Press P again to create the polygon (with 3 or more points)
Try it now with the 3 markers given

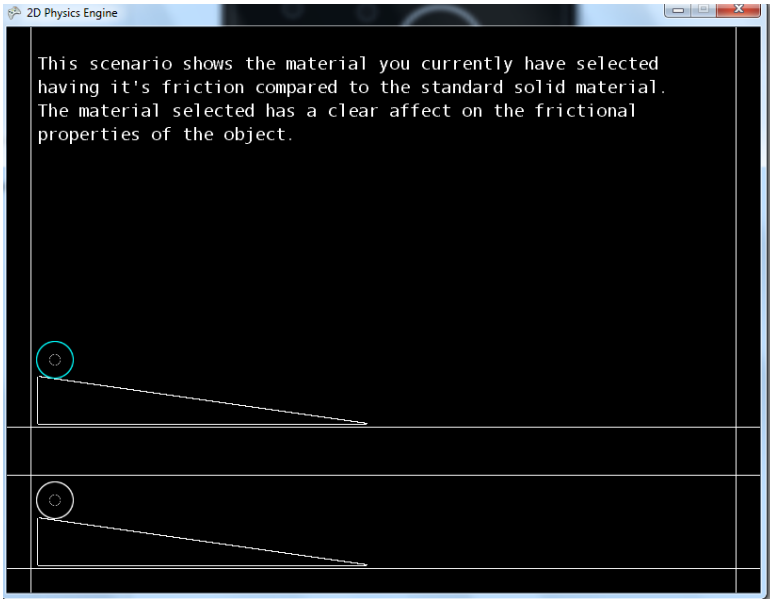

2D Physics Engine
This is the circle drawing tutorial. The controls are as below:
-Press O to initiate polygon drawing mode
-Click and Drag to set radius and position (Start of drag is centre)
-Press O again to create the circle
Try it now with the 2 markers given

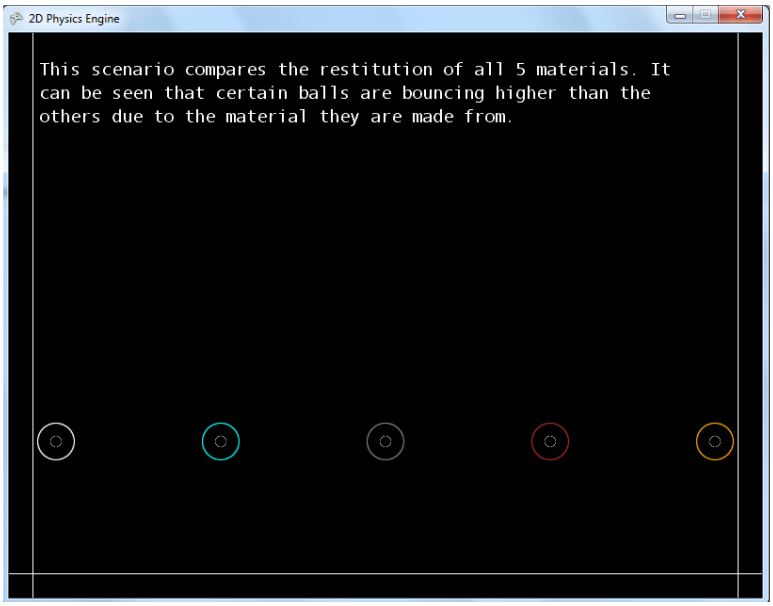

2D Physics Engine
In this scenario, the ball gathers speed as it rolls down the ramp.
It then collides with the other ball which is at rest
The collision causes the 1st ball to slow down and the second ball
to move. Thereby conserving the overall momentum and kinetic energy
of the system.


2D Physics Engine
This scenario compares how far two different sized balls move when an equal impulse is applied. The balls rolling down the ramps end with the same amount of kinetic energy. The larger ball moves with less speed in order to conserve momentum it also exerts a greater impulse on the ball that hits it.


2D Physics Engine
This scenario shows how a ball of equal size but greater speed exerts a greater impulse. The ball on top travels down a steeper gradient and so gathers more speed. This causes the ball it hits to move with greater speed and travel further.


2D Physics Engine
This scenario shows off the zero gravity affect, all but one of the balls is unaffected by gravity. It also shows off the engine's ability to deal with a large amount of collisions.


2D Physics Engine
This scenario shows the material you currently have selected having it's friction compared to the standard solid material. The material selected has a clear affect on the frictional properties of the object.


2D Physics Engine
This scenario compares the restitution of all 5 materials. It can be seen that certain balls are bouncing higher than the others due to the material they are made from.

ProgramManager Code and Explanation

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;

using NEA_Physics_Engine.Physics;
using NEA_Physics_Engine.Physics.Properties;
using NEA_Physics_Engine.Rendering;
using NEA_Physics_Engine.Utilities;
using NEA_Physics_Engine.Input;

namespace NEA_Physics_Engine
{
        ///Main type for the engine
        public class ProgramManager : Microsoft.Xna.Framework.Game
        {
    //Attributes
    //SCREEN_WIDTH (integer): the screen width in pixels
    //SCREEN_HEIGHT (integer): the screen height in pixels
    //my_Shape (Texture2D): the texture of shapes
    //texture_Background (Texture2D): the texture of the background
    //my_DeltaTime (Float): a float used to keep track of time
    //lastUpdate (Double): a double used to track when the last update occured
    //lastDraw (Double): a double used to track when the last object was drawn
    //drawingCircle (Bool): a boolean used to track whether the circle drawing
mode is engaged
    //drawingPolygon (Bool): a boolean used to track whether the polygon
drawing mode is engaged
    //showControls (Bool): a boolean used to decide whether control help
should be displayed or not
    //isStatic (Bool): a boolean used to decide whether the next object drawn
should be affected by collisions and gravity
    //gravityAffects (Bool): a boolean used to decide whether the next object
drawn is affected by gravity
    //MaterialOfNextObject (Integer): an integer which corresponds to what
material the next object drawn will be made of
    //preset (Integer): an integer which corresponds to the preset on display
    //circleRadius (Float): a float which contains the radius of the circle being
drawn
```

```
    //circlePosition (Vector2): a vector which contains the central position of the circle being drawn
    //polygonVertices (List of Vector2): a list of vectors which give the co-ordinates of each vertex of the polygon being drawn

    //Methods
    //Initialize: an override of the basic visual studio initialize, used so managers can be initialised
    //drawPreset1: draws the 1st preset
    //drawPreset2: draws the 2nd preset
    //drawPreset3: draws the 3rd preset
    //drawPreset4: draws the 4th preset
    //drawPreset5: draws the 5th preset
    //drawPreset6: draws the 6th preset
    //drawPreset7: draws the 7th preset
    //drawPreset8: draws the 8th preset
    //drawPreset9: draws the 9th preset
    //drawPreset10: draws the 10th preset
    //LoadContent: draws the sandbox mode and is run at the start
    //UnloadContent: unloads the content
    //Update: runs on update, controls the drawing tool and calls other managers to update
    //DrawResults: Draws results of the update
    //DeltaTime: Get my_DeltaTime
    //ScreenWidth: Get SCREEN_WIDTH
    //ScreenHeight: Get SCREEN_HEIGHT

    //Height and Width in pixels
            private const int SCREEN_WIDTH = 800;
            private const int SCREEN_HEIGHT = 600;



    //Textures for shapes and the background
    private Texture2D my_Shape;
            public Texture2D texture_Background;

    //Reference to other scripts in the base engine namespace
            private ObjectsManager my_ObjectsManager;
            private PhysicsManager my_PhysicsManager;
    private InputManager my_InputManager;
    private RenderManager my_RenderManager;



    //Initialise time
            private float my_DeltaTime = 0;
```

```
    //Setup the graphics device manager which handles config. and
management of the graphics device
                public GraphicsDeviceManager my_Graphics {get; private set;}

                ///Allows engine to perform initialisations it requires before it
runs.
                ///Can get required services and load any related content.
                protected override void Initialize()
                {
                        my_ObjectsManager = ObjectsManager.Instance;
        my_PhysicsManager = PhysicsManager.Instance;
        my_RenderManager = RenderManager.Instance;
                        my_InputManager = InputManager.Instance;
                        base.Initialize();
                }

    //-----------------------------------**Preset Drawing Functions**----------------------
-----------------------------------
    public void drawPreset1()
    {
        my_ObjectsManager.Clear();
        //Scene content, set out as a series of vertices
        Vector2[] vertices = new Vector2[4];
        vertices[3] = new Vector2(-SCREEN_WIDTH * 0.5f, 50);
        vertices[2] = new Vector2(SCREEN_WIDTH * 0.5f, 50);
        vertices[1] = new Vector2(SCREEN_HEIGHT * 0.5f, -200);
        vertices[0] = new Vector2(-SCREEN_HEIGHT * 0.5f, -200);
        Vector2[] vertices2 = new Vector2[4];
        vertices2[3] = new Vector2(0, SCREEN_HEIGHT);
        vertices2[2] = new Vector2(50, SCREEN_HEIGHT);
        vertices2[1] = new Vector2(50, 0);
        vertices2[0] = new Vector2(0, 0);
        Vector2[] vertices3 = new Vector2[4];
        vertices3[3] = new Vector2(0, 50);
        vertices3[2] = new Vector2(SCREEN_WIDTH, 50);
        vertices3[1] = new Vector2(SCREEN_WIDTH, 0);
        vertices3[0] = new Vector2(0, 0);
        Vector2[] vertices4 = new Vector2[3];
        vertices4[2] = new Vector2(-(SCREEN_WIDTH * 0.5f) + 50, 50);
        vertices4[1] = new Vector2(-(SCREEN_WIDTH * 0.5f) + 50, 100);
        vertices4[0] = new Vector2(0, 100);

        //Adds scenes base polygons i.e. walls and floor
        my_ObjectsManager.Add(new GameObject(new Vector2(0,
SCREEN_HEIGHT * 0.5f), my_Shape, new PhysicsPolygonDefine(vertices2,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2(SCREEN_WIDTH,
SCREEN_HEIGHT * 0.5f), my_Shape, new PhysicsPolygonDefine(vertices2,
true)));
```

```
        my_ObjectsManager.Add(new GameObject(new Vector2(SCREEN_WIDTH
* 0.5f, SCREEN_HEIGHT), my_Shape, new PhysicsPolygonDefine(vertices3,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 250, SCREEN_HEIGHT - 45), my_Shape, new
PhysicsPolygonDefine(vertices4, true)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 350, SCREEN_HEIGHT - 100), my_Shape, new PhysicsCircleDefine(20,
false)));
    }

    public void drawPreset2()
    {
        my_ObjectsManager.Clear();
        //Scene content, set out as a series of vertices
        Vector2[] vertices = new Vector2[4];
        vertices[3] = new Vector2(-SCREEN_WIDTH * 0.5f, 50);
        vertices[2] = new Vector2(SCREEN_WIDTH * 0.5f, 50);
        vertices[1] = new Vector2(SCREEN_HEIGHT * 0.5f, -200);
        vertices[0] = new Vector2(-SCREEN_HEIGHT * 0.5f, -200);
        Vector2[] vertices2 = new Vector2[4];
        vertices2[3] = new Vector2(0, SCREEN_HEIGHT);
        vertices2[2] = new Vector2(50, SCREEN_HEIGHT);
        vertices2[1] = new Vector2(50, 0);
        vertices2[0] = new Vector2(0, 0);
        Vector2[] vertices3 = new Vector2[4];
        vertices3[3] = new Vector2(0, 50);
        vertices3[2] = new Vector2(SCREEN_WIDTH, 50);
        vertices3[1] = new Vector2(SCREEN_WIDTH, 0);
        vertices3[0] = new Vector2(0, 0);
        Vector2[] vertices4 = new Vector2[6];
        vertices4[5] = new Vector2(SCREEN_WIDTH * 0.5f, 50);
        vertices4[4] = new Vector2(20, 90);
        vertices4[3] = new Vector2(-20, 90);
        vertices4[2] = new Vector2(-(SCREEN_WIDTH * 0.5f) + 50, 50);
        vertices4[1] = new Vector2(-(SCREEN_WIDTH * 0.5f) + 50, 100);
        vertices4[0] = new Vector2(SCREEN_WIDTH * 0.5f, 100);

        //Adds scenes base polygons i.e. walls and floor
        my_ObjectsManager.Add(new GameObject(new Vector2(0,
SCREEN_HEIGHT * 0.5f), my_Shape, new PhysicsPolygonDefine(vertices2,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2(SCREEN_WIDTH,
SCREEN_HEIGHT * 0.5f), my_Shape, new PhysicsPolygonDefine(vertices2,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2(SCREEN_WIDTH
* 0.5f, SCREEN_HEIGHT), my_Shape, new PhysicsPolygonDefine(vertices3,
true)));
```

```
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) + 7, SCREEN_HEIGHT - 45), my_Shape, new
PhysicsPolygonDefine(vertices4, true)));
        my_ObjectsManager.Add(new GameObject(new
Vector2((SCREEN_WIDTH) - 100, SCREEN_HEIGHT - 100), my_Shape, new
PhysicsCircleDefine(20, false)));
    }

    public void drawPreset5()
    {
        my_ObjectsManager.Clear();
        //Scene content, set out as a series of vertices
        Vector2[] vertices = new Vector2[4];
        vertices[3] = new Vector2(-SCREEN_WIDTH * 0.5f, 50);
        vertices[2] = new Vector2(SCREEN_WIDTH * 0.5f, 50);
        vertices[1] = new Vector2(SCREEN_HEIGHT * 0.5f, -200);
        vertices[0] = new Vector2(-SCREEN_HEIGHT * 0.5f, -200);
        Vector2[] vertices2 = new Vector2[4];
        vertices2[3] = new Vector2(0, SCREEN_HEIGHT);
        vertices2[2] = new Vector2(50, SCREEN_HEIGHT);
        vertices2[1] = new Vector2(50, 0);
        vertices2[0] = new Vector2(0, 0);
        Vector2[] vertices3 = new Vector2[4];
        vertices3[3] = new Vector2(0, 50);
        vertices3[2] = new Vector2(SCREEN_WIDTH, 50);
        vertices3[1] = new Vector2(SCREEN_WIDTH, 0);
        vertices3[0] = new Vector2(0, 0);
        Vector2[] vertices4 = new Vector2[3];
        vertices4[2] = new Vector2(-(SCREEN_WIDTH * 0.5f) + 50, 50);
        vertices4[1] = new Vector2(-(SCREEN_WIDTH * 0.5f) + 50, 100);
        vertices4[0] = new Vector2(0, 100);

        //Adds scenes base polygons i.e. walls and floor
        my_ObjectsManager.Add(new GameObject(new Vector2(0,
SCREEN_HEIGHT * 0.5f), my_Shape, new PhysicsPolygonDefine(vertices2,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2(SCREEN_WIDTH,
SCREEN_HEIGHT * 0.5f), my_Shape, new PhysicsPolygonDefine(vertices2,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2(SCREEN_WIDTH
* 0.5f, SCREEN_HEIGHT), my_Shape, new PhysicsPolygonDefine(vertices3,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 250, SCREEN_HEIGHT - 45), my_Shape, new
PhysicsPolygonDefine(vertices4, true)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 350, SCREEN_HEIGHT - 100), my_Shape, new PhysicsCircleDefine(20,
false)));
```

```
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) + 150, SCREEN_HEIGHT - 70), my_Shape, new PhysicsCircleDefine(20,
false)));
    }

    public void drawPreset6()
    {
        my_ObjectsManager.Clear();
        //Scene content, set out as a series of vertices
        Vector2[] vertices = new Vector2[4];
        vertices[3] = new Vector2(-SCREEN_WIDTH * 0.5f, 50);
        vertices[2] = new Vector2(SCREEN_WIDTH * 0.5f, 50);
        vertices[1] = new Vector2(SCREEN_HEIGHT * 0.5f, -200);
        vertices[0] = new Vector2(-SCREEN_HEIGHT * 0.5f, -200);
        Vector2[] vertices2 = new Vector2[4];
        vertices2[3] = new Vector2(0, SCREEN_HEIGHT);
        vertices2[2] = new Vector2(50, SCREEN_HEIGHT);
        vertices2[1] = new Vector2(50, 0);
        vertices2[0] = new Vector2(0, 0);
        Vector2[] vertices3 = new Vector2[4];
        vertices3[3] = new Vector2(0, 50);
        vertices3[2] = new Vector2(SCREEN_WIDTH, 50);
        vertices3[1] = new Vector2(SCREEN_WIDTH, 0);
        vertices3[0] = new Vector2(0, 0);
        Vector2[] vertices4 = new Vector2[3];
        vertices4[2] = new Vector2(-(SCREEN_WIDTH * 0.5f) + 50, 50);
        vertices4[1] = new Vector2(-(SCREEN_WIDTH * 0.5f) + 50, 100);
        vertices4[0] = new Vector2(0, 100);

        //Adds scenes base polygons i.e. walls and floor
        my_ObjectsManager.Add(new GameObject(new Vector2(0,
SCREEN_HEIGHT * 0.5f), my_Shape, new PhysicsPolygonDefine(vertices2,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2(SCREEN_WIDTH,
SCREEN_HEIGHT * 0.5f), my_Shape, new PhysicsPolygonDefine(vertices2,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2(SCREEN_WIDTH
* 0.5f, SCREEN_HEIGHT), my_Shape, new PhysicsPolygonDefine(vertices3,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2(SCREEN_WIDTH
* 0.5f, SCREEN_HEIGHT * 0.75f), my_Shape, new
PhysicsPolygonDefine(vertices3, true)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 250, SCREEN_HEIGHT - 45), my_Shape, new
PhysicsPolygonDefine(vertices4, true)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 250, (SCREEN_HEIGHT * 0.75f) - 45), my_Shape, new
PhysicsPolygonDefine(vertices4, true)));
```

```
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 350, SCREEN_HEIGHT - 100), my_Shape, new PhysicsCircleDefine(20,
false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) + 150, SCREEN_HEIGHT - 70), my_Shape, new PhysicsCircleDefine(20,
false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 350, (SCREEN_HEIGHT * 0.75f) - 100), my_Shape, new
PhysicsCircleDefine(20, false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) + 150, (SCREEN_HEIGHT * 0.75f) - 70), my_Shape, new
PhysicsCircleDefine(40, false)));
    }

    public void drawPreset7()
    {
        my_ObjectsManager.Clear();
        //Scene content, set out as a series of vertices
        Vector2[] vertices = new Vector2[4];
        vertices[3] = new Vector2(-SCREEN_WIDTH * 0.5f, 50);
        vertices[2] = new Vector2(SCREEN_WIDTH * 0.5f, 50);
        vertices[1] = new Vector2(SCREEN_HEIGHT * 0.5f, -200);
        vertices[0] = new Vector2(-SCREEN_HEIGHT * 0.5f, -200);
        Vector2[] vertices2 = new Vector2[4];
        vertices2[3] = new Vector2(0, SCREEN_HEIGHT);
        vertices2[2] = new Vector2(50, SCREEN_HEIGHT);
        vertices2[1] = new Vector2(50, 0);
        vertices2[0] = new Vector2(0, 0);
        Vector2[] vertices3 = new Vector2[4];
        vertices3[3] = new Vector2(0, 50);
        vertices3[2] = new Vector2(SCREEN_WIDTH, 50);
        vertices3[1] = new Vector2(SCREEN_WIDTH, 0);
        vertices3[0] = new Vector2(0, 0);
        Vector2[] vertices4 = new Vector2[3];
        vertices4[2] = new Vector2(-(SCREEN_WIDTH * 0.5f) + 50, 50);
        vertices4[1] = new Vector2(-(SCREEN_WIDTH * 0.5f) + 50, 100);
        vertices4[0] = new Vector2(0, 100);
        Vector2[] vertices5 = new Vector2[3];
        vertices5[2] = new Vector2(-(SCREEN_WIDTH * 0.5f) + 50, 0);
        vertices5[1] = new Vector2(-(SCREEN_WIDTH * 0.5f) + 50, 100);
        vertices5[0] = new Vector2(0, 100);

        //Adds scenes base polygons i.e. walls and floor
        my_ObjectsManager.Add(new GameObject(new Vector2(0,
SCREEN_HEIGHT * 0.5f), my_Shape, new PhysicsPolygonDefine(vertices2,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2(SCREEN_WIDTH,
SCREEN_HEIGHT * 0.5f), my_Shape, new PhysicsPolygonDefine(vertices2,
true)));
```

```
        my_ObjectsManager.Add(new GameObject(new Vector2(SCREEN_WIDTH
* 0.5f, SCREEN_HEIGHT), my_Shape, new PhysicsPolygonDefine(vertices3,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2(SCREEN_WIDTH
* 0.5f, SCREEN_HEIGHT * 0.75f), my_Shape, new
PhysicsPolygonDefine(vertices3, true)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 250, SCREEN_HEIGHT - 45), my_Shape, new
PhysicsPolygonDefine(vertices4, true)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 250, (SCREEN_HEIGHT * 0.75f) - 63), my_Shape, new
PhysicsPolygonDefine(vertices5, true)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 350, SCREEN_HEIGHT - 100), my_Shape, new PhysicsCircleDefine(20,
false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) + 150, SCREEN_HEIGHT - 70), my_Shape, new PhysicsCircleDefine(20,
false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 350, (SCREEN_HEIGHT * 0.75f) - 150), my_Shape, new
PhysicsCircleDefine(20, false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) + 150, (SCREEN_HEIGHT * 0.75f) - 70), my_Shape, new
PhysicsCircleDefine(20, false)));
    }

    protected void drawPreset8()
    {
        my_ObjectsManager.Clear();



        //Scene content, set out as a series of vertices
        Vector2[] vertices = new Vector2[4];
        vertices[3] = new Vector2(-SCREEN_WIDTH * 0.5f, 50);
        vertices[2] = new Vector2(SCREEN_WIDTH * 0.5f, 50);
        vertices[1] = new Vector2(SCREEN_HEIGHT * 0.5f, -200);
        vertices[0] = new Vector2(-SCREEN_HEIGHT * 0.5f, -200);
        Vector2[] vertices2 = new Vector2[4];
        vertices2[3] = new Vector2(0, SCREEN_HEIGHT);
        vertices2[2] = new Vector2(50, SCREEN_HEIGHT);
        vertices2[1] = new Vector2(50, 0);
        vertices2[0] = new Vector2(0, 0);
        Vector2[] vertices3 = new Vector2[4];
        vertices3[3] = new Vector2(0, 50);
        vertices3[2] = new Vector2(SCREEN_WIDTH, 50);
        vertices3[1] = new Vector2(SCREEN_WIDTH, 0);
        vertices3[0] = new Vector2(0, 0);
```

```
        //Adds scenes base polygons i.e. walls and floor
        my_ObjectsManager.Add(new GameObject(new Vector2(0,
SCREEN_HEIGHT * 0.5f), my_Shape, new PhysicsPolygonDefine(vertices2,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2(SCREEN_WIDTH,
SCREEN_HEIGHT * 0.5f), my_Shape, new PhysicsPolygonDefine(vertices2,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2(SCREEN_WIDTH
* 0.5f, SCREEN_HEIGHT), my_Shape, new PhysicsPolygonDefine(vertices3,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 350, (SCREEN_HEIGHT * 0.75f) - 150), my_Shape, new
PhysicsCircleDefine(20, false, Material.SOLID, false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) + 150, (SCREEN_HEIGHT * 0.75f) - 70), my_Shape, new
PhysicsCircleDefine(20, false, Material.SOLID, false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) + 180, (SCREEN_HEIGHT * 0.75f) - 272), my_Shape, new
PhysicsCircleDefine(20, false, Material.SOLID, false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 342, (SCREEN_HEIGHT * 0.75f) - 73), my_Shape, new
PhysicsCircleDefine(20, false, Material.SOLID, false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 325, (SCREEN_HEIGHT * 0.75f) - 124), my_Shape, new
PhysicsCircleDefine(20, false, Material.SOLID, false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) + 157, (SCREEN_HEIGHT * 0.75f) - 62), my_Shape, new
PhysicsCircleDefine(20, false, Material.SOLID, false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 200, (SCREEN_HEIGHT * 0.75f) - 74), my_Shape, new
PhysicsCircleDefine(20, false, Material.SOLID, false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 220, (SCREEN_HEIGHT * 0.75f) - 85), my_Shape, new
PhysicsCircleDefine(20, false, Material.SOLID, false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 210, (SCREEN_HEIGHT * 0.75f) - 25), my_Shape, new
PhysicsCircleDefine(20, false, Material.SOLID, false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 320, (SCREEN_HEIGHT * 0.75f) - 74), my_Shape, new
PhysicsCircleDefine(20, false, Material.SOLID, false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) + 132, (SCREEN_HEIGHT * 0.75f) - 70), my_Shape, new
PhysicsCircleDefine(20, false, Material.SOLID, false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) + 143, (SCREEN_HEIGHT * 0.75f) - 272), my_Shape, new
PhysicsCircleDefine(20, false, Material.SOLID, false)));
```

```
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 250, (SCREEN_HEIGHT * 0.75f) - 57), my_Shape, new
PhysicsCircleDefine(20, false, Material.SOLID, false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 320, (SCREEN_HEIGHT * 0.75f) - 145), my_Shape, new
PhysicsCircleDefine(20, false, Material.SOLID, false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) + 165, (SCREEN_HEIGHT * 0.75f) - 58), my_Shape, new
PhysicsCircleDefine(20, false, Material.SOLID, false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 210, (SCREEN_HEIGHT * 0.75f) - 71), my_Shape, new
PhysicsCircleDefine(20, false, Material.SOLID, false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 223, (SCREEN_HEIGHT * 0.75f) - 88), my_Shape, new
PhysicsCircleDefine(20, false, Material.SOLID, false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 243, (SCREEN_HEIGHT * 0.75f) - 35), my_Shape, new
PhysicsCircleDefine(20, false, Material.SOLID, false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f), (SCREEN_HEIGHT * 0.75f)), my_Shape, new PhysicsCircleDefine(20, false,
Material.SOLID, false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 100, (SCREEN_HEIGHT * 0.75f) - 300), my_Shape, new
PhysicsCircleDefine(100, false)));
    }

    public void drawPreset9()
    {
        my_ObjectsManager.Clear();
        //Scene content, set out as a series of vertices
        Vector2[] vertices = new Vector2[4];
        vertices[3] = new Vector2(-SCREEN_WIDTH * 0.5f, 50);
        vertices[2] = new Vector2(SCREEN_WIDTH * 0.5f, 50);
        vertices[1] = new Vector2(SCREEN_HEIGHT * 0.5f, -200);
        vertices[0] = new Vector2(-SCREEN_HEIGHT * 0.5f, -200);
        Vector2[] vertices2 = new Vector2[4];
        vertices2[3] = new Vector2(0, SCREEN_HEIGHT);
        vertices2[2] = new Vector2(50, SCREEN_HEIGHT);
        vertices2[1] = new Vector2(50, 0);
        vertices2[0] = new Vector2(0, 0);
        Vector2[] vertices3 = new Vector2[4];
        vertices3[3] = new Vector2(0, 50);
        vertices3[2] = new Vector2(SCREEN_WIDTH, 50);
        vertices3[1] = new Vector2(SCREEN_WIDTH, 0);
        vertices3[0] = new Vector2(0, 0);
        Vector2[] vertices4 = new Vector2[3];
        vertices4[2] = new Vector2(-(SCREEN_WIDTH * 0.5f) + 50, 50);
        vertices4[1] = new Vector2(-(SCREEN_WIDTH * 0.5f) + 50, 100);
        vertices4[0] = new Vector2(0, 100);
```

```
        //Adds scenes base polygons i.e. walls and floor
        my_ObjectsManager.Add(new GameObject(new Vector2(0,
SCREEN_HEIGHT * 0.5f), my_Shape, new PhysicsPolygonDefine(vertices2,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2(SCREEN_WIDTH,
SCREEN_HEIGHT * 0.5f), my_Shape, new PhysicsPolygonDefine(vertices2,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2(SCREEN_WIDTH
* 0.5f, SCREEN_HEIGHT), my_Shape, new PhysicsPolygonDefine(vertices3,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2(SCREEN_WIDTH
* 0.5f, SCREEN_HEIGHT * 0.75f), my_Shape, new
PhysicsPolygonDefine(vertices3, true)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 250, SCREEN_HEIGHT - 45), my_Shape, new
PhysicsPolygonDefine(vertices4, true)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 250, (SCREEN_HEIGHT * 0.75f) - 45), my_Shape, new
PhysicsPolygonDefine(vertices4, true)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 350, SCREEN_HEIGHT - 100), my_Shape, new PhysicsCircleDefine(20,
false)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 350, (SCREEN_HEIGHT * 0.75f) - 100), my_Shape, new
PhysicsCircleDefine(20, false, MaterialOfNextObject)));
    }

    protected void drawPreset10()
    {
        my_ObjectsManager.Clear();



        //Scene content, set out as a series of vertices
        Vector2[] vertices = new Vector2[4];
        vertices[3] = new Vector2(-SCREEN_WIDTH * 0.5f, 50);
        vertices[2] = new Vector2(SCREEN_WIDTH * 0.5f, 50);
        vertices[1] = new Vector2(SCREEN_HEIGHT * 0.5f, -200);
        vertices[0] = new Vector2(-SCREEN_HEIGHT * 0.5f, -200);
        Vector2[] vertices2 = new Vector2[4];
        vertices2[3] = new Vector2(0, SCREEN_HEIGHT);
        vertices2[2] = new Vector2(50, SCREEN_HEIGHT);
        vertices2[1] = new Vector2(50, 0);
        vertices2[0] = new Vector2(0, 0);
        Vector2[] vertices3 = new Vector2[4];
        vertices3[3] = new Vector2(0, 50);
        vertices3[2] = new Vector2(SCREEN_WIDTH, 50);
        vertices3[1] = new Vector2(SCREEN_WIDTH, 0);
```

```
        vertices3[0] = new Vector2(0, 0);


        //Adds scenes base polygons i.e. walls and floor
        my_ObjectsManager.Add(new GameObject(new Vector2(0,
SCREEN_HEIGHT * 0.5f), my_Shape, new PhysicsPolygonDefine(vertices2,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2(SCREEN_WIDTH,
SCREEN_HEIGHT * 0.5f), my_Shape, new PhysicsPolygonDefine(vertices2,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2(SCREEN_WIDTH
* 0.5f, SCREEN_HEIGHT), my_Shape, new PhysicsPolygonDefine(vertices3,
true)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f), (SCREEN_HEIGHT * 0.75f) - 100), my_Shape, new PhysicsCircleDefine(20,
false, Material.METAL)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 175, (SCREEN_HEIGHT * 0.75f) - 100), my_Shape, new
PhysicsCircleDefine(20, false, Material.ICE)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) - 350, (SCREEN_HEIGHT * 0.75f) - 100), my_Shape, new
PhysicsCircleDefine(20, false, Material.SOLID)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) + 175, (SCREEN_HEIGHT * 0.75f) - 100), my_Shape, new
PhysicsCircleDefine(20, false, Material.WOOD)));
        my_ObjectsManager.Add(new GameObject(new Vector2((SCREEN_WIDTH
* 0.5f) + 350, (SCREEN_HEIGHT * 0.75f) - 100), my_Shape, new
PhysicsCircleDefine(20, false, Material.RUBBER)));
    }

    //--------------------------------------**End of Preset Drawing Functions**---------
---------------------------------------

                //LoadContent is called once and loads all the content
                protected override void LoadContent()
                {
        my_ObjectsManager.Clear();
        texture_Background = Content.Load<Texture2D>("background");


        //Scene content, set out as a series of vertices
                        Vector2[] vertices = new Vector2[4];
                        vertices[3] = new Vector2(-SCREEN_WIDTH * 0.5f, 50);
                        vertices[2] = new Vector2(SCREEN_WIDTH * 0.5f, 50);
                        vertices[1] = new Vector2(SCREEN_HEIGHT * 0.5f, -200);
                        vertices[0] = new Vector2(-SCREEN_HEIGHT * 0.5f, -200);
                        Vector2[] vertices2 = new Vector2[4];
                        vertices2[3] = new Vector2(0, SCREEN_HEIGHT);
                        vertices2[2] = new Vector2(50, SCREEN_HEIGHT);
```

```
                        vertices2[1] = new Vector2(50, 0);
                        vertices2[0] = new Vector2(0, 0);
                        Vector2[] vertices3 = new Vector2[4];
                        vertices3[3] = new Vector2(0, 50);
                        vertices3[2] = new Vector2(SCREEN_WIDTH, 50);
                        vertices3[1] = new Vector2(SCREEN_WIDTH, 0);
                        vertices3[0] = new Vector2(0, 0);


        //Adds scenes base polygons i.e. walls and floor
                        my_ObjectsManager.Add(new GameObject(new Vector2(0,
SCREEN_HEIGHT * 0.5f), my_Shape, new PhysicsPolygonDefine(vertices2,
true)));
                        my_ObjectsManager.Add(new GameObject(new
Vector2(SCREEN_WIDTH, SCREEN_HEIGHT * 0.5f), my_Shape, new
PhysicsPolygonDefine(vertices2, true)));
                        my_ObjectsManager.Add(new GameObject(new
Vector2(SCREEN_WIDTH * 0.5f, SCREEN_HEIGHT), my_Shape, new
PhysicsPolygonDefine(vertices3, true)));
                }

                ///UnloadContents will be called once and unloads all the content
                protected override void UnloadContent()
                {
                        Content.Unload();
                }

                ///Updates the scene and the physics
                private double lastUpdate = 0;
                private double lastDraw = 0;
                bool drawingCircle = false;
                bool drawingPolygon = false;
    bool showControls = true;
                bool isStatic = false;
    bool gravityAffects = true;
    int MaterialOfNextObject = 1;
    int preset = 0;
                float circleRadius;
                Vector2 circleStartPosition;
                List<Vector2> polygonVertices = new List<Vector2>();

                protected override void Update(GameTime runTime)
                {
        //Updates time
                        my_DeltaTime =
(float)runTime.ElapsedGameTime.TotalSeconds;

                        ///Allow engine to exit
                        if (Keyboard.GetState().IsKeyDown(Keys.Escape))
```

```
                this.Exit();


                double endtime = runTime.TotalGameTime.Milliseconds;

        //----------------------------------------**Checking for presets**----------
--------------------------------

        //Haven't used switch statements (Case Statements) because they require
jump statements between cases
        if (preset == 1)
        {
            my_RenderManager.DrawString("  This scenario shows a ball rolling
down the ramp demonstrating");
            my_RenderManager.DrawString("  gravity and collision resolution.");
            my_RenderManager.DrawString("  If you watch the ball after it comes off
the ramp, you'll");
            my_RenderManager.DrawString("  notice it stops just short of the
wall.");
            my_RenderManager.DrawString("  This is due to a combination of
damping and frictional force.");
        }

        if (preset == 2)
        {
            my_RenderManager.DrawString("  If you watch this scenario you'll see
that the ball gains speed");
            my_RenderManager.DrawString("  as it rolls down the first side of the
ramp but it does not");
            my_RenderManager.DrawString("  gain enough kinetic energy to roll all
the way up the other");
            my_RenderManager.DrawString("  side and thus rolls back down.");
        }

        if (preset == 3)
        {
            my_RenderManager.DrawString("  This is the polygon drawing tutorial.
The controls are as below:");
            my_RenderManager.DrawString("  -Press P to initiate polygon drawing
mode");
            my_RenderManager.DrawString("  -Click to place a point (if cursor is red
it means the point is");
            my_RenderManager.DrawString("   invalid.");
            my_RenderManager.DrawString("  -Press P again to create the polygon
(with 3 or more points)");
            my_RenderManager.DrawString(" ");
            my_RenderManager.DrawString("  Try it now with the 3 markers
given");
```

```
            my_RenderManager.DrawCircle(new Vector2(SCREEN_WIDTH * 0.5f,
SCREEN_HEIGHT - 300), 10, Color.Yellow);
            my_RenderManager.DrawCircle(new Vector2((SCREEN_WIDTH * 0.5f) +
100, SCREEN_HEIGHT - 350), 10, Color.Yellow);
            my_RenderManager.DrawCircle(new Vector2((SCREEN_WIDTH * 0.5f) +
200, SCREEN_HEIGHT - 300), 10, Color.Yellow);
        }

        if (preset == 4)
        {
            my_RenderManager.DrawString("  This is the circle drawing tutorial.
The controls are as below:");
            my_RenderManager.DrawString("  -Press O to initiate polygon drawing
mode");
            my_RenderManager.DrawString("  -Click and Drag to set radius and
position (Start of drag is centre)");
            my_RenderManager.DrawString("  -Press O again to create the circle");
            my_RenderManager.DrawString(" ");
            my_RenderManager.DrawString("  Try it now with the 2 markers
given");
            my_RenderManager.DrawCircle(new Vector2(SCREEN_WIDTH * 0.5f,
SCREEN_HEIGHT - 300), 10, Color.Yellow);
            my_RenderManager.DrawCircle(new Vector2(SCREEN_WIDTH * 0.5f,
SCREEN_HEIGHT - 300), 40, Color.Yellow);
        }
        if (preset == 5)
        {
            my_RenderManager.DrawString("  In this scenario, the ball gathers
speed as it rolls down the ramp.");
            my_RenderManager.DrawString("  It then collides with the other ball
which is at rest");
            my_RenderManager.DrawString("  The collision causes the 1st ball to
slow down and the second ball");
            my_RenderManager.DrawString("  to move. Thereby conserving the
overall momentum and kinetic energy");
            my_RenderManager.DrawString("  of the system.");
        }
        if (preset == 6)
        {
            my_RenderManager.DrawString("  This scenario compares how far two
different sized balls move when");
            my_RenderManager.DrawString("  an equal impulse is applied. The balls
rolling down the ramps");
            my_RenderManager.DrawString("  end with the same amount of kinetic
energy. The larger ball moves");
            my_RenderManager.DrawString("  with less speed in order to conserve
momentum it also exerts a ");
            my_RenderManager.DrawString("  greater impulse on the ball that hits
it.");
```

```
        }
        if (preset == 7)
        {
            my_RenderManager.DrawString("  This scenario shows how a ball of
equal size but greater speed");
            my_RenderManager.DrawString("  exerts a greater impulse. The ball on
top travels down a");
            my_RenderManager.DrawString("  steeper gradient and so gathers more
speed. This causes the ball");
            my_RenderManager.DrawString("  it hits to move with greater speed
and travel further.");
        }
        if (preset == 8)
        {
            my_RenderManager.DrawString("  This scenario shows off the zero
gravity affect, all but one");
            my_RenderManager.DrawString("  of the balls is unaffected by gravity. It
also shows off the");
            my_RenderManager.DrawString("  engine's ability to deal with a large
amount of collisions.");
        }
        if (preset == 9)
        {
            my_RenderManager.DrawString("  This scenario shows the material you
currently have selected");
            my_RenderManager.DrawString("  having it's friction compared to the
standard solid material.");
            my_RenderManager.DrawString("  The material selected has a clear
affect on the frictional");
            my_RenderManager.DrawString("  properties of the object.");
        }
        if (preset == 10)
        {
            my_RenderManager.DrawString("  This scenario compares the
restitution of all 5 materials. It");
            my_RenderManager.DrawString("  can be seen that certain balls are
bouncing higher than the ");
            my_RenderManager.DrawString("  others due to the material they are
made from.");
        }
        //----------------------------------------**End of preset checks**----------
-------------------------------

        //---------------------------------------**Miscellaneous Input checks**-------
-----------------------------------

        //Clear objects when user inputs C
        if (my_InputManager.KeyWasPressed(Keys.C))
        {
```

```
            my_ObjectsManager.Clear();
        }

        //Decide whether new object will be static (i.e. it feels no force but has
collision physics)
        if (my_InputManager.KeyWasPressed(Keys.Q))
        {
            if (isStatic)
                isStatic = false;
            else
                isStatic = true;
        }

        //Decide whether new object will be affected by gravity (automatically
false when static is true)
        if (my_InputManager.KeyWasPressed(Keys.G))
        {
            if (gravityAffects)
                gravityAffects = false;
            else
                gravityAffects = true;
        }

        if (isStatic)
            gravityAffects = false;

        //------------------------**Load Preset Inputs**-------------------------------
        if (my_InputManager.KeyWasPressed(Keys.D0))
        {
            LoadContent();
            preset = 0;
        }

        if (my_InputManager.KeyWasPressed(Keys.D1))
        {
            showControls = false;
            drawPreset1();
            preset = 1;
        }

        if (my_InputManager.KeyWasPressed(Keys.D2))
        {
            showControls = false;
            drawPreset2();
            preset = 2;
        }

        if (my_InputManager.KeyWasPressed(Keys.D3))
        {
```

```
    showControls = false;
    LoadContent();
    preset = 3;
}

if (my_InputManager.KeyWasPressed(Keys.D4))
{
    showControls = false;
    LoadContent();
    preset = 4;
}

if (my_InputManager.KeyWasPressed(Keys.D5))
{
    showControls = false;
    drawPreset5();
    preset = 5;
}

if (my_InputManager.KeyWasPressed(Keys.D6))
{
    showControls = false;
    drawPreset6();
    preset = 6;
}

if (my_InputManager.KeyWasPressed(Keys.D7))
{
    showControls = false;
    drawPreset7();
    preset = 7;
}

if (my_InputManager.KeyWasPressed(Keys.D8))
{
    showControls = false;
    drawPreset8();
    preset = 8;
}

if (my_InputManager.KeyWasPressed(Keys.D9))
{
    showControls = false;
    drawPreset9();
    preset = 9;
}
if (my_InputManager.KeyWasPressed(Keys.F1))
{
    showControls = false;
```

```
            drawPreset10();
            preset = 10;
        }
        //------------------------**End of Load Preset Inputs**-----------------------------
-

        //------------------------**Material switching inputs**-----------------------------
        if (my_InputManager.KeyWasPressed(Keys.NumPad1))
        {
            MaterialOfNextObject = 1;
        }

        if (my_InputManager.KeyWasPressed(Keys.NumPad2))
            MaterialOfNextObject = 2;

        if (my_InputManager.KeyWasPressed(Keys.NumPad3))
            MaterialOfNextObject = 3;

        if (my_InputManager.KeyWasPressed(Keys.NumPad4))
            MaterialOfNextObject = 4;

        if (my_InputManager.KeyWasPressed(Keys.NumPad5))
            MaterialOfNextObject = 5;
        //------------------------**End of Material switching inputs**----------------------
---------
        //----------------------------------**End of miscallaneous input checks**----------
-------------------------------------

        if (preset != 0)
        {
            //Lock gravity manipulation during preset scenarios
            my_PhysicsManager.Gravity = new Vector2 (0f, 20.0f);
        }

        if ((preset != 0) && (preset != 3))
        {
            drawingPolygon = false;
        }

        if ((preset != 0) && (preset != 4))
        {
            drawingCircle = false;
        }

                        if (drawingPolygon == true || drawingCircle == true)
                        {
            //tells user whether the object will be static, whether gravity is applied
(if non-static) and what material it is made of
```

```
            RenderManager.Instance.DrawString("   Gravity: " +
gravityAffects.ToString());
                        RenderManager.Instance.DrawString("   Static: " +
isStatic.ToString());
                        RenderManager.Instance.DrawString("   Material: " +
MaterialOfNextObject.ToString());
                }

        //Toggle polygon drawing mode when P is entered
                if (my_InputManager.KeyWasPressed(Keys.P))
                {
                        if (drawingPolygon == false)
                                drawingPolygon = true;
                        else if (polygonVertices.Count > 2)
                        {
            //Creates polygon out of points given by the user
                                Vector2 startPosition = new Vector2();

            //Calculate center point
                                foreach (Vector2 vertex in polygonVertices)
                                        startPosition += vertex;
                                startPosition.X /= polygonVertices.Count;
                                startPosition.Y /= polygonVertices.Count;

            //Add object to the list of objects
                                my_ObjectsManager.Add(new
GameObject(startPosition, my_Shape, new
PhysicsPolygonDefine(polygonVertices.ToArray(), isStatic,
MaterialOfNextObject, gravityAffects)));

            //Clear the vertices list of points for this polygon
                                polygonVertices.Clear();
                                drawingPolygon = false;
                        }
                        else
                        {
            //If there aren't enough points to draw more than a line (i.e. if there
are less than 3 points)...
            //then clear the vertices list of points for this polygon
                                drawingPolygon = false;
                                polygonVertices.Clear();
                        }
                }
        //Toggle circle drawing mode when O is entered
                else if (my_InputManager.KeyWasPressed(Keys.O))
                {
                        if (drawingCircle == false)
                                drawingCircle = true;
                        else if (circleStartPosition != Vector2.Zero)
```

```
                                {
            //When circle's position is in a valid position i.e. not vector (0,0), add
to the objects list
            my_ObjectsManager.Add(new GameObject(circleStartPosition,
my_Shape, new PhysicsCircleDefine(circleRadius, isStatic, MaterialOfNextObject,
gravityAffects)));

            //Reset the circle properties ready for the next circle
                                        circleStartPosition = Vector2.Zero;
                                        circleRadius = 0;
                                        drawingCircle = false;
                                }
                                else
                                        drawingCircle = false;
                        }
        //Toggle control help when X is entered
        else if (my_InputManager.KeyWasPressed(Keys.X))
        {
          if (showControls == false)
            showControls = true;
          else
            showControls = false;
        }

                        if (drawingPolygon)
                        {
          //Tell user they are in polygon drawing mode
                                RenderManager.Instance.DrawString("   Drawing
polygon");
                                foreach (Vector2 vertex in polygonVertices)
                                {
            //Create small circles to represent the vertices (will use to show
collision detection)
                                        RenderManager.Instance.DrawCircle(vertex,
6);
                                }
                                if (polygonVertices.Count > 2)
                                {
            //Find perpendicular vectors for line between first 2 vertices,
between current vertex and last vertex, between current vertex and first vertex
                                        Vector2 normal1 =
mathsUtility.leftPerpendicular(polygonVertices[polygonVertices.Count - 1] -
polygonVertices[polygonVertices.Count - 2]);
                                        Vector2 normal2 =
mathsUtility.leftPerpendicular(polygonVertices[1] - polygonVertices[0]);
                                        Vector2 normal3 =
mathsUtility.leftPerpendicular(polygonVertices[0] -
polygonVertices[polygonVertices.Count - 1]);
                                        normal1.Normalize();
```

```
                                        normal2.Normalize();
            //Create edges
                                        Vector2 newEdge1 =
polygonVertices[polygonVertices.Count - 1] -
my_InputManager.GetMousePosition();
                                        Vector2 newEdge2 = polygonVertices[0] -
my_InputManager.GetMousePosition();

            //Check normals and edges aren't going to intersect
                                        if (Vector2.Dot(newEdge1, normal1) < 0 &&
Vector2.Dot(newEdge2, normal2) < 0 && Vector2.Dot(newEdge2, normal3) > 0)
                                        {
                //If new edge is valid draw a green point on the mouse position to
indicate vertex can be placed

        RenderManager.Instance.DrawCircle(my_InputManager.GetMousePositio
n(), 6, Color.Green);
                                            if
(my_InputManager.LeftMouseWasPressed())
                    //Add vertex to the list for the current polygon

        polygonVertices.Add(my_InputManager.GetMousePosition());
                                        }
                                        else
                //If new edge is invalid draw a red point on the mouse position to
indicate vertex can't be placed

        RenderManager.Instance.DrawCircle(my_InputManager.GetMousePositio
n(), 6, Color.Red);
                                    }
                                    else
                                    {
                                        if (polygonVertices.Count == 2)
                                        {
                //Since only on the second vertex there are only two points and thus
one edge with one normal
                                            Vector2 edge =
my_InputManager.GetMousePosition() - polygonVertices[1];
                                            Vector2 normal =
mathsUtility.leftPerpendicular(polygonVertices[1] - polygonVertices[0]);
                                            normal.Normalize();
                                            if (Vector2.Dot(edge, normal) > 0)
                                            {
                    //If vertex in valid position, draw green point to indicate vertex
can be placed

        RenderManager.Instance.DrawCircle(my_InputManager.GetMousePositio
n(), 6, Color.Green);
```

```
                //Add vertex on click
                                                    if
(my_InputManager.LeftMouseWasPressed())

        polygonVertices.Add(my_InputManager.GetMousePosition());
                                                    }
                                                    else
                //If position is invalid, draw red point to indicate vertex can't be
placed

        RenderManager.Instance.DrawCircle(my_InputManager.GetMousePositio
n(), 6, Color.Red);
                                            }
                                            else
                                            {
                //If first point it can be placed anywhere so point is always green

        RenderManager.Instance.DrawCircle(my_InputManager.GetMousePositio
n(), 6, Color.Green);

                //Add vertex on click
                                                    if
(my_InputManager.LeftMouseWasPressed())

        polygonVertices.Add(my_InputManager.GetMousePosition());
                                                    }
                                            }
                                    }
                                    else if (drawingCircle)
                                    {
         //Notify user they are in circle drawing mode
                                            RenderManager.Instance.DrawString("   Drawing
circle");

         //Set start position of circle on click
                                            if (my_InputManager.LeftMouseWasPressed())
                                                    circleStartPosition =
my_InputManager.GetMousePosition();

         //Set circle radius when clicked again
                                            if (my_InputManager.LeftMouseIsPressed())
                                                    circleRadius = (circleStartPosition -
my_InputManager.GetMousePosition()).Length();


        RenderManager.Instance.DrawCircle(circleStartPosition, circleRadius,
Color.Green);
                                    }
```

```
        //Control help display
        if (showControls == true)
        {
            RenderManager.Instance.DrawString("   P : Toggle Polygon Drawing
Mode||O : Toggle Circle Drawing Mode");
            RenderManager.Instance.DrawString("   X : Toggle Control Help||Q :
Toggle Static");
            RenderManager.Instance.DrawString("   G : Toggle Whether Gravity
Affects Next Object");
            RenderManager.Instance.DrawString("   M : Toggle Selected Object
Information");
            RenderManager.Instance.DrawString("   NUMPAD 1 : Material 1
(Solid)||NUMPAD 2 : Material 2 (Ice)");
            RenderManager.Instance.DrawString("   NUMPAD 3 : Material 3
(Wood)||NUMPAD 4 : Material 4 (Metal)");
            RenderManager.Instance.DrawString("   NUMPAD 5 : Material 5
(Rubber)");
            RenderManager.Instance.DrawString("   W : Change Gravity Vector Up||S
: Change Gravity Vector Down");
            RenderManager.Instance.DrawString("   A : Change Gravity Vector
Left||D : Change Gravity Vector Right");
            RenderManager.Instance.DrawString("   0 : Sandbox||1 : Scenario 1||2 :
Scenario 2||3 : Scenario 3");
            RenderManager.Instance.DrawString("   4 : Scenario 4||5 : Scenario 5||6 :
Scenario 6");
            RenderManager.Instance.DrawString("   7 : Scenario 7||8 : Scenario 8||9 :
Scenario 9");
            RenderManager.Instance.DrawString("   F1 : Scenario 10");

            if ((drawingPolygon == false) & (drawingCircle == false))
            {
                RenderManager.Instance.DrawString("   Left Click : Select Object");
            }
            if (drawingPolygon == true)
            {
                RenderManager.Instance.DrawString("   Left Click : Place Vertex");
            }
            if (drawingCircle == true)
            {
                RenderManager.Instance.DrawString("   Left Click and Drag : Set
Radius");
            }
        }

        //Update other managers
                        my_PhysicsManager.Update();
                        my_ObjectsManager.Update();
                        my_InputManager.Update();
        my_RenderManager.Update();
```

```
                base.Update(runTime);
            }

            //This is called when the engine should draw the results of the
update
            protected void DrawResults(GameTime runTime)
            {
                //Updates the renderer making it draw

        NEA_Physics_Engine.Rendering.RenderManager.Instance.Update();

                double endtime = runTime.TotalGameTime.Milliseconds -
lastDraw;
                lastDraw = runTime.TotalGameTime.Milliseconds;

                base.Draw(runTime);
            }

    //Get my_DeltaTime
            public float DeltaTime {get {return my_DeltaTime;}}

    //Get window properties
            public int ScreenHeight {get {return SCREEN_HEIGHT;}}
            public int ScreenWidth {get {return SCREEN_WIDTH;}}

    //Set up instance of program manager
            private static volatile ProgramManager my_Instance;
            private static object my_SyncRoot = new Object();
            private ProgramManager()
            {
      //Setup window title, buffer and root directory of content
                my_Graphics = new GraphicsDeviceManager(this);
                Window.Title = "2D Physics Engine";
                my_Graphics.PreferredBackBufferWidth = SCREEN_WIDTH;
                my_Graphics.PreferredBackBufferHeight =
SCREEN_HEIGHT;
                my_Graphics.ApplyChanges();
                Content.RootDirectory = "Content";

      //Make the mouse visible to aid in drawing
                IsMouseVisible = true;
                my_Graphics.SynchronizeWithVerticalRetrace = true;
                IsFixedTimeStep = false;
            }

    //Run instance of program manager
            public static ProgramManager Instance
            {
```

```
                get
                {
                        if (my_Instance == null)
                                lock (my_SyncRoot)
                        if (my_Instance == null)
                        {
                                my_Instance = new ProgramManager();
                        }
                        return my_Instance;
                }
        }
    }
}
```

The program manager acts as the central file of the program and is in charge of calling updates and setting up the basics. It also houses the code for the drawing tools and draws presets. It also makes some changes based on user input normally to do with showing controls, changing materials, etc. Whilst most of the code in here is miscellaneous checks and setting up shapes, there is one block of code I'd like to talk about: the validation for polygon drawing.

The polygons need to be convex and be represented in a clockwise manner. The code (highlighted in yellow) makes sure this is always true. It starts off by checking how many points are already in place. If more than two points are in place then it stores 3 normal vectors, these are perpendicular to the edges formed by the last 2 vectors (1st normal), the first 2 vectors (2nd normal) and the last vector and the first vector (3rd normal). It then stores 2 new edges, one from the last vertex to where the user is trying to place a point (1st edge) and another from the first vertex to where the user is trying to place a point (2nd edge). It then uses the vector dot product to check what kind of angle it's going to make with the normals, if the angle the new edges make with the first two normals is obtuse and the second edge makes an acute angle with the third normal then the point will result in a convex shape and is therefore valid. The next case is that there are only two points, here only one vertex and one edge need to be considered. As long as the edge and normal make an acute angle then it's valid. The final case is when there are one or less points, in this case any point is valid. The reason the checks makes sure the points are represented clockwise is because of the use of the left perpendicular as opposed to the right perpendicular (these would just be negatives of one another).

ObjectsManager Code and Explanation

```
using System;
using System.Collections.Generic;

using NEA_Physics_Engine.Interfaces;
using NEA_Physics_Engine.Physics;

namespace NEA_Physics_Engine
{
        class ObjectsManager : Interfaces.Updateable
        {
    //Attributes
    //my_Objects (List of GameObject): a list of the game objects that currently
exist

    //Methods
    //Add: add a new object to the list
    //Clear: clear the list
    //Update: run on each update to update each object

    //Create list that contains all game objects
            private static List<GameObject> my_Objects;

    //Will add new object to the game objects list
            public void Add(GameObject new_PhysicsObject)
            {
                    my_Objects.Add(new_PhysicsObject);
            }

    //Clears all objects from the screen and removes their details from the
program
            public void Clear()
            {
                    my_Objects.Clear();
                    PhysicsManager.Instance.Clear();
            }

    //Update each object
            public void Update()
            {
                    foreach (GameObject shape in my_Objects)
                            shape.Update();
            }

            private static volatile ObjectsManager my_Instance;
            private static object my_SyncRoot = new Object();
```

```
        private ObjectsManager()
        {
            my_Objects = new List<GameObject>();
        }

    //Run instance of objects manager
        public static ObjectsManager Instance
        {
            get
            {
                if (my_Instance == null)
                    lock (my_SyncRoot)

                if (my_Instance == null)
                {
                    my_Instance = new ObjectsManager();
                }

            return my_Instance;
            }
        }
    }
}
```

The objects manager is in charge of maintaining the game objects list. As such it has very few methods. It has an add method for adding new game objects, a clear method for emptying the list and an update method for calling each game object's update method in the list.

GameObject Code and Explanation

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

using NEA_Physics_Engine.Interfaces;
using NEA_Physics_Engine.Physics;
using NEA_Physics_Engine.Rendering;
using NEA_Physics_Engine.Physics.Properties;

namespace NEA_Physics_Engine
{
        public sealed class GameObject : Interfaces.Updateable
        {
    //Attributes
    //my_Body (PhysicsBody): the corresponding physics body to this game
object
    //my_Texture (Texture2D): the texture of the object
    //my_Colour (Color): the colour of the object

    //Methods
    //Position: get and set position
    //Body: get my_Body
    //Update: run on every update to redraw the object

    //Create variables for the object being dealt with, the texture of the object
and the colour of the object.
                private PhysicsBody my_Body;
                private Texture2D my_Texture;
                private Color my_Colour;

    //Get/Set the position of the object
                public Vector2 Position
                {
                        get
                        {
                                return my_Body.Position;
                        }
                        private set
                        {
                                my_Body.Position = value;
                        }
                }

    //Get my_Body
                public PhysicsBody Body {get {return my_Body;}}
```

```
    //Initialisation
            public GameObject(Vector2 parameter_Position, Texture2D
parameter_Texture, PhysicsShape parameter_ShapeDefine)
            {
        //get texture and object
                    my_Texture = parameter_Texture;
                    my_Body = new PhysicsBody(parameter_Position,
parameter_ShapeDefine);

        //Switch colour based on material (case not used because they require
jump statements)
        if (parameter_ShapeDefine.Material == 1)
          my_Colour = Color.White;
        if (parameter_ShapeDefine.Material == 2)
          my_Colour = Color.Cyan;
        if (parameter_ShapeDefine.Material == 3)
          my_Colour = Color.Brown;
        if (parameter_ShapeDefine.Material == 4)
          my_Colour = Color.Gray;
        if (parameter_ShapeDefine.Material == 5)
          my_Colour = Color.Orange;
            }


            public void Update()
            {
        //Have the render manager draw the object, with the given properties.
                    if (Body.Shape == Shape.CIRCLE)
                        RenderManager.Instance.DrawCircle(Position,
Body.CircleDefine.Radius, my_Colour);
                    else
                    {
          //Polygons drawn as a set of lines
                        for (int i = 0; i < Body.PolygonDefine.VerticesCount;
i++)
                            RenderManager.Instance.DrawLine(Position
+ Body.PolygonDefine.GetVertex(i), Position + Body.PolygonDefine.GetVertex((i
+ 1 == Body.PolygonDefine.VerticesCount ? 0 : i + 1)), 1, my_Colour);
                    }
            }
        }
}
```

The game object class is used for setting up game objects (i.e. polygons and circles). When it is instantiated it sets the object's texture based on a parameter and instantiates a physics body which corresponds to itself. It then assigns itself a colour based on its material. It also contains an update function which redraws the object at its new location.

InputManager Code and Explanation

```
using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Input;

using NEA_Physics_Engine.Interfaces;

namespace NEA_Physics_Engine.Input
{
        class InputManager : Interfaces.Updateable
        {
    //Attributes
    //my_PreviousKeyboardState (KeyboardState): stores previous state of the
keyboard
    //my_PreviousMouseState (MouseState): stores previous state of the mouse

    //Methods
    //Update: runs on update and updates the keyboard and mouse states
    //GetMousePosition: returns the mouse's current position
    //LeftMouseWasPressed: returns whether left mouse button was pressed
between last update and this update
    //LeftMouseIsPressed: returns whether left mouse button is currently
pressed
    //LeftMouseIsReleased: returns whether left mouse button is currently
released
    //KeyWasPressed: returns whether a particular key was pressed between
last update and this update
    //KeyIsPressed: returns whether a particular key is currently pressed
    //KeysPressed: returns keys that are currently pressed

    //Variables to get the state of the keyboard and mouse when last run
                KeyboardState my_PreviousKeyboardState;
                MouseState my_PreviousMouseState;

                public virtual void Update()
                {
      //On update set the state in the previous update to previous state
                        my_PreviousKeyboardState = Keyboard.GetState();
                        my_PreviousMouseState = Mouse.GetState();
                }

    //Get a position of the cursor
                public Vector2 GetMousePosition()
                {
                        return new Vector2(Mouse.GetState().X,
Mouse.GetState().Y);
```

```
		}

	//Determine whether the left mouse button was pressed since the last
update
		public bool LeftMouseWasPressed()
		{
			return Mouse.GetState().LeftButton == ButtonState.Pressed
&& my_PreviousMouseState.LeftButton != ButtonState.Pressed;
		}

	//Determine if the left mouse button is currently pressed
		public bool LeftMouseIsPressed()
		{
			return Mouse.GetState().LeftButton ==
ButtonState.Pressed;
		}

	//Determine if left mouse is released
	public bool LeftMouseIsReleased()
	{
	  return Mouse.GetState().LeftButton != ButtonState.Pressed;
	}

	//Determine whether a key was pressed since the last update
	public bool KeyWasPressed(Keys key)
	{
	  return Keyboard.GetState().IsKeyDown(key) &&
my_PreviousKeyboardState.IsKeyUp(key);
	}

	//Determine if a key is currently pressed
	public bool KeyIsPressed(Keys key)
	{
	  return Keyboard.GetState().IsKeyDown(key);
	}

	//Determine which keys are currently pressed
	public Keys[] KeysPressed()
	{
	  return Keyboard.GetState().GetPressedKeys();
	}

		private static volatile InputManager my_Instance;
		private static object my_SyncRoot = new Object();

	//Create an instance of the input manager
		public static InputManager Instance
		{
			get
```

```
                {
                        if (my_Instance == null)
                                lock (my_SyncRoot)

                        if (my_Instance == null)
                        {
                                my_Instance = new InputManager();
                        }

                        return my_Instance;
                }

        }
    }
}
```

The input manager monitors the state of the keyboard and the state of the mouse and has methods the checking the current states and any changes in state. These methods are normally used as Booleans for conditional statements.

Updateable Code and Explanation

```
namespace NEA_Physics_Engine.Interfaces
{
  //Creates an interface that can be updated
        interface Updateable
        {
                void Update();
        }
}
```

It just provides an interface for the classes to interact with. It's required but just has an update method.

PhysicsBody Code and Explanation

```
using System;

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Input;

using NEA_Physics_Engine.Physics.Properties;
using NEA_Physics_Engine.Rendering;
using NEA_Physics_Engine.Interfaces;
using NEA_Physics_Engine.Input;
using NEA_Physics_Engine.Utilities;

namespace NEA_Physics_Engine.Physics
{
  //Keep store of types of force
        public enum FORCE_TYPE
        {
                IMPULSE,
                FORCE
        };

        public class PhysicsBody : Interfaces.Updateable
        {
    //Attributes
    //my_ObjectCount (integer): the number of objects
    //ShapeDefine (PhysicsShape): the instance of PhysicsShape linked to this
physics body
    //ShapeCircleDefine (PhysicsCircleDefine): the instance of
PhysicsCircleDefine linked to this physics body, if appropriate
    //ShapePolygonDefine (PhysicsPolygonDefine): the instance of
PhysicsPolygonDefine linked to this physics body, if appropriate
    //my_ID (Integer): an ID number unique to this object
    //my_Force (Vector2): a vector containing the x and y components of the
force applied
```

```
    //my_Acceleration (Vector2): a vector containing the x and y components of the acceleration of the object
    //my_LinearVelocity (Vector2): a vector containing the x and y components of the linear velocity
    //my_AngularVelocity (Float): a float containg the angular velocity of the object
    //my_Position (Vector2): a vector containing the x and y co-ordinates of the object's position
    //my_LastPosition (Vector2): a vector containing the x and y co-ordinates of the object's position at the previous update
    //my_InverseMass (Float): the value of 1/Mass
    //my_InverseInertia (Float): the value of 1/Inertia

    //Methods
    //PhysicsBody: defines properties about the object
    //Update: run on update to update position and other properties
    //GetID: returns the objects ID
    //isStatic: returns whether object is static
    //AddForce: adds a given force or impulse to the object
    //AddTorque: adds a given amount of torque to the object
    //Position: get/set my_Position
    //LastPosition: get/set my_LastPosition
    //Force: get/set my_Force
    //LinearVelocity: get/set my_LinearVelocity
    //AngularVelocity: get/set my_AngularVelocity
    //Acceleration: get/set my_Accekeration
    //Shape: get my_Shape
    //Material: get my_Material
    //Restitution: get/set restitution coefficient
    //StaticFriction: get/set static coefficient of friction
    //DynamicFriction: get/set dynamic coefficient of friction
    //Mass: get/set mass of object
    //Volume: get/set volume of object
    //InverseMass: get my_InverseMass
    //Inertia: get/set inertia of object
    //InverseInertia: get my_InverseInertia

    //Used to keep count of the number of objects
            private static int my_ObjectCount = 0;

    //Defined so they can be used to define new shapes
            public PhysicsShape ShapeDefine = null;
            public PhysicsCircleDefine CircleDefine = null;
            public PhysicsPolygonDefine PolygonDefine = null;

    //An ID for the object
            private int my_ID;
```

```
    //Set some empty vectors that can be used to store the force, acceleration,
velocity, current and previous position
                private Vector2 my_Force = new Vector2();
                private Vector2 my_Acceleration = new Vector2();
                private Vector2 my_LinearVelocity = new Vector2();
    private float my_AngularVelocity = 0;
                private Vector2 my_Position = new Vector2();
                private Vector2 my_LastPosition = new Vector2();

    //Inverse mass and inertia to be used in calculations
                private float my_InverseMass = -1;
    private float my_InverseInertia = -1;



                public PhysicsBody(Vector2 parameter_Position, PhysicsShape
parameter_ShapeDefine)
                {
        //Get position and definition of the current shape
                        ShapeDefine = parameter_ShapeDefine;
                        Position = parameter_Position;
        float Restitution = parameter_ShapeDefine.Restitution;
        float StaticFriction = parameter_ShapeDefine.StaticFriction;
        float DynamicFriction = parameter_ShapeDefine.DynamicFriction;

        //Get the shape's mass
                        Mass = parameter_ShapeDefine.Mass;

        //Set the shape's ID
                        my_ID = my_ObjectCount++;

        //Define the shape dependent on the shape property of the shape
        if (parameter_ShapeDefine.Shape == 1)
        {
          CircleDefine = (PhysicsCircleDefine)parameter_ShapeDefine;
        }
        else if (parameter_ShapeDefine.Shape == 2)
        {
          PolygonDefine = (PhysicsPolygonDefine)parameter_ShapeDefine;
        }

        //Add the object to the list
                        PhysicsManager.Instance.AddBody(this);
                }

                public virtual void Update()
                {
        //Store position before update
                        LastPosition = Position;
```

```
        //Get the change in time
                        float deltaTime = ProgramManager.Instance.DeltaTime;

        //Update angle
        ShapeDefine.Rotate(0.2f * angularVelocity * deltaTime);

        //Calculate acceleration using re-arrangement of F=ma, a = F/m. (Use
inverse mass in case the mass is 0)
                        Acceleration = Force * InverseMass;

        //Calculate velocity using delta v = a * delta t
                        linearVelocity += Acceleration * deltaTime;

        //Damping
        linearVelocity *= 0.99f;
        angularVelocity *= 0.995f;

        //Calculate position using delta s = v * delta t
                        Position += (linearVelocity * deltaTime);

                        if (isStatic() == false)
                                RenderManager.Instance.DrawCircle(Position, 6);


        //Apply the force at the position
                        AddForce(-Force, Position);
                }

    //Get the object's ID
    public virtual int GetID() { return my_ID; }

    //If shape is static then set mass to 0
                public bool isStatic() {return ShapeDefine.Mass == 0;}

                public void AddForce(Vector2 parameter_Force, Vector2
parameter_Position, FORCE_TYPE parameter_ForceType = FORCE_TYPE.FORCE)
                {
        //Two types of force, a force or an impulse. In general impulse is used but
force is there in case certain scenarios are required
        if (parameter_ForceType == FORCE_TYPE.FORCE)
           Force += parameter_Force;
        else if (parameter_ForceType == FORCE_TYPE.IMPULSE)
           linearVelocity += parameter_Force * InverseMass;


                        if (parameter_Position == Position)
                                return;
```

```
            //Calculate torque
            //Radius-vector from center of mass to position force being applied
            Vector2 radiusVector = parameter_Position - Position;
            float torque = mathsUtility.CrossProduct(radiusVector,
parameter_Force);
            //Apply torque generated by the force acting on the body
            AddTorque(torque);
                  }


      //Add the torque of the force
      public void AddTorque(float parameter_Torque)
      {
         angularVelocity += (parameter_Torque * InverseInertia);
      }

      //Get/Set position
                  public Vector2 Position
                  {
                        get {return my_Position;}
                        set {my_Position = value;}
                  }

      //Get/Set previous position
                  public Vector2 LastPosition
                  {
                        get {return my_LastPosition;}
                        private set {my_LastPosition = value;}
                  }

      //Get/Set forces applied
                  public Vector2 Force
                  {
                        get {return my_Force;}
                        private set {my_Force = value;}
                  }

      //Get/Set linear velocity
                  public Vector2 linearVelocity
                  {
                        get {return my_LinearVelocity;}
                        private set {my_LinearVelocity = value;}
                  }


      //Get/Set angular velocity
      public float angularVelocity
      {
         get { return my_AngularVelocity; }
```

```
        private set
        {
            my_AngularVelocity = value;
            //Limit to avoid insanity
            if (Math.Abs(my_AngularVelocity) >
Limits.MAXIMUM_ANGULAR_VELOCITY)
                if (my_AngularVelocity > 0)
                    my_AngularVelocity = Limits.MAXIMUM_ANGULAR_VELOCITY;
                else
                    my_AngularVelocity = -Limits.MAXIMUM_ANGULAR_VELOCITY;
            else if (Math.Abs(my_AngularVelocity) <
Limits.MINIMUM_ANGULAR_VELOCITY)
                my_AngularVelocity = 0;
        }
    }

    //Get/Set acceleration
                public Vector2 Acceleration
                {
                        get {return my_Acceleration;}
                        private set {my_Acceleration = value;}
                }

    //Get/Set type of shape
                public int Shape
                {
                        get {return ShapeDefine.Shape;}
                }

    //Get/Set material
                public int Material
                {
                        get {return ShapeDefine.Material;}
                }

    //Get/Set restitution
    public float Restitution
    {
        get { return ShapeDefine.Restitution; }
        set { ShapeDefine.Restitution = value; }
    }

    //Get/Set static friction
    public float StaticFriction
    {
        get { return ShapeDefine.StaticFriction; }
        set { ShapeDefine.StaticFriction = value; }
    }
```

```
//Get/Set dynamic friction
public float DynamicFriction
{
   get { return ShapeDefine.DynamicFriction; }
   set { ShapeDefine.DynamicFriction = value; }
}

//Get/Set mass and calculate the inverse mass to be used in calcs.
            public float Mass
            {
                  get {return ShapeDefine.Mass;}
                  private set
                  {
                        ShapeDefine.Mass = value;
                        if (ShapeDefine.Mass > 0)
                              my_InverseMass = 1 / ShapeDefine.Mass;
                        else
                              my_InverseMass = 0;
                  }
            }

public float Volume
{
   get { return ShapeDefine.Volume; }
   set { }
}

//Get inverse mass
            public float InverseMass
            {
                  get {return my_InverseMass;}
            }


//Get/Set Inertia
public float Inertia
{
   get { return ShapeDefine.Inertia; }
   private set
   {
      ShapeDefine.Inertia = value;
      if (ShapeDefine.Inertia > 0)
         my_InverseInertia = 1 / ShapeDefine.Inertia;
      else
         my_InverseInertia = 0;
   }
}

//Get inverse inertia
```

```
        public float InverseInertia
        {
            get { return my_InverseInertia; }
        }
            }
}
```

The physics body class is mainly used for keeping track of the physical properties of a body and allowing other classes to get and set them. It also updates the physical properties and is used to add any external force and the equivalent torque.

The main physical updates are highlighted in blue. It gets the change in time and then updates properties accordingly. We can apply

*Distance = Speed * Time*

to angles to get

*Angle = Angular Speed * Time*

and using this we can change the angle of the shape (Note: 0.2 is an arbitrary scalar). We can also reorganise

*Force = Mass * Acceleration*

to give a formula for acceleration

*Acceleration = Force / Mass = Force * Inverse Mass*

and use this to update the acceleration property. The linear velocity is then updated using

*Velocity = Initial Velocity + (Acceleration * Time)*

Damping is then applied (a change caused by air resistance and some other resistances). Damping is very complex and there are different models for representing how it works. A very common method is linear damping, which is what is used here. The linear velocity and angular velocity need to be damped. Finally the position is updated using

*Displacement = Initial Displacement + (Linear Velocity * Time)*

The other method to really talk about is the AddForce method (highlighted in yellow). There are two type of force allowed by the engine: a force or an impulse. A force affects acceleration and is constant and an impulse is a force which acts for a short time frame (like kicking a ball). When a force is applied then the Force attribute is changed. If an impulse is applied then

*Linear Velocity = Initial Velocity + (Impulse / Mass) = Initial Velocity + (Impulse * Inverse Mass)*

and this then is applied to the linearVelocity property. Every force exerts a certain amount of torque on an object. So the torque is calculated in the AddForce method. Torque in two dimensions is given by

Torque = Force x Radius

As such a radius vector is needed to cross with the force parameter that was given. The radius vector is given by the position vector of where the force was applied minus the centre of the object. Once these have been passed into the CrossProduct method found in the mathsUtility class, the torque can be applied.

PhysicsShape Code and Explanation

```
using Microsoft.Xna.Framework;

namespace NEA_Physics_Engine.Physics
{
        public abstract class PhysicsShape
        {
    //Attributes
    //my_AABB (AABB): contains data about the axis-aligned bounding box of the object

    //Methods
    //Shape: get/set the shape of the object
    //Material: get/set the material of the object
    //Density: get/set the density of the object
    //Angle: get/set the angle of the object
    //Volume: get/set the volume of the object
    //Restitution: get/set the restitution coefficient of the object
    //StaticFriction: get/set the static coefficient of friction of the object
    //DynamicFriction: get/set the dynamic coefficient of friction of the object
    //Mass: get/set the mass of the object
    //Inertia: get/set the inertia of the object
    //Gravity: get/set whether the object is affected by gravity
    //SetAABB: initialise data about the axis-aligned bounding box
    //CalculateAABB: an abstract method which shares a name with methods in PhysicsShapeCircle and PhysicsShapePolygon
    //CalculateVolume: an abstract method which shares a name with methods in PhysicsShapeCircle and PhysicsShapePolygon
    //CalculateMass: an abstract method which shares a name with methods in PhysicsShapeCircle and PhysicsShapePolygon
    //CalculateCenterOfMass: an abstract method which shares a name with methods in PhysicsShapeCircle and PhysicsShapePolygon
```

```
    //ComputeInertia: an abstract method which shares a name with methods in
PhysicsShapeCircle and PhysicsShapePolygon

    //Layout the structure of the axis aligned bounding box
            public struct AABB
            {
      //Using the center and half the height and half the width, we can
determine all points we need for the bounding box
                    public Vector2 Center {get; set;}
                    public float HalfWidth {get; set;}
                    public float HalfHeight {get; set;}
            }

    //Get shape properties
            private AABB my_AABB;
            public int Shape {get; protected set;}
            public int Material {get; protected set;}
            public int Density {get; set;}
    public float Angle { get; protected set; }
            public float Volume {get; protected set;}
    public float Restitution { get; set; }
    public float StaticFriction { get; set; }
    public float DynamicFriction { get; set;}
            public float Mass {get; set;}
    public float Inertia { get; set; }
    public bool Gravity { get; set; }

    //Set up the axis aligned bounding box
            protected void SetAABB(float parameter_HalfWidth, float
parameter_HalfHeight)
            {
                    my_AABB.Center = new Vector2();
                    my_AABB.HalfWidth = parameter_HalfWidth;
                    my_AABB.HalfHeight = parameter_HalfHeight;
            }

    //Calculate needed properties of the shape
            public AABB GetAABB() {return my_AABB;}
    public abstract void Rotate(float angle, Vector2 originTranslation = new
Vector2());
            protected abstract void CalculateAABB();
            protected abstract void CalculateVolume();
            protected abstract void CalculateMass();
            protected abstract void CalculateCenterOfMass();
    protected abstract void ComputeInertia();
      }
}
```

The physics shape class is an abstract class and is used to essentially provide a blueprint of what subclasses of physics shape should include. So all the abstract methods should be included as override methods in the subclass. This means that if anyone wants to add new types of shapes in the future then they won't be able to miss out any integral methods.

PhysicsShapeCircle Code and Explanation

```
using System;
using Microsoft.Xna.Framework;

using NEA_Physics_Engine.Physics.Properties;
using NEA_Physics_Engine.Utilities;

namespace NEA_Physics_Engine.Physics
{

        public sealed class PhysicsCircleDefine : PhysicsShape
        {
    //Methods
    //PhysicsCircleDefine: used to define properties of the circle
    //Rotate: used to rotate the object
    //Radius: get/set the radius of the circle
    //CalculateAABB: used to set data about the axis-aligned bounding box
    //CalculateMass: used to calculate the mass of the circle
    //ComputeInertia: used to calculate the inertia of the circle
    //CalculateCentreOfMass: used to calculate the centre of mass of the circle
    //CalculateVolume: used to calculate the volume of the circle via a redirect
to the mathsUtility

    //Get the properties of the circle
                public PhysicsCircleDefine(PhysicsCircleDefine other)
                {
                        Shape = other.Shape;
      Mass = other.Mass;
                        Radius = other.Radius;
      Restitution = other.Restitution;
      StaticFriction = other.StaticFriction;
      DynamicFriction = other.DynamicFriction;
      Inertia = other.Inertia;
      Gravity = other.Gravity;
                }

                public PhysicsCircleDefine(float parameter_Radius, bool isStatic =
false, int parameter_Material = Properties.Material.SOLID, bool gravityAffects =
true)
                {
      //Set properties of the circle that aren't given
                        Material = parameter_Material;
```

```
                    Radius = parameter_Radius;

        Gravity = gravityAffects;

                    Shape = 1;

        //Restitution and Friction changes based on material, case not used
because requires a jump statement
        if (Material == 1)
        {
           Restitution = Properties.Restitution.SOLID;
           StaticFriction = Properties.StaticFriction.SOLID;
           DynamicFriction = Properties.DynamicFriction.SOLID;
           Density = Properties.Density.SOLID;
        }
        if (Material == 2)
        {
           Restitution = Properties.Restitution.ICE;
           StaticFriction = Properties.StaticFriction.ICE;
           DynamicFriction = Properties.DynamicFriction.ICE;
           Density = Properties.Density.ICE;
        }
        if (Material == 3)
        {
           Restitution = Properties.Restitution.WOOD;
           StaticFriction = Properties.StaticFriction.WOOD;
           DynamicFriction = Properties.DynamicFriction.WOOD;
           Density = Properties.Density.WOOD;
        }
        if (Material == 4)
        {
           Restitution = Properties.Restitution.METAL;
           StaticFriction = Properties.StaticFriction.METAL;
           DynamicFriction = Properties.DynamicFriction.METAL;
           Density = Properties.Density.METAL;
        }
        if (Material == 5)
        {
           Restitution = Properties.Restitution.RUBBER;
           StaticFriction = Properties.StaticFriction.RUBBER;
           DynamicFriction = Properties.DynamicFriction.RUBBER;
           Density = Properties.Density.RUBBER;
        }

        //Calculate the volume and the bounding box
                    CalculateVolume();
                    CalculateAABB();
        ComputeInertia();
```

```
        //Calculate mass using density and volume, unless object is static (in
which case there is no mass, so no effect by gravity)
                        if (isStatic == false)
                        {
                                CalculateMass();
                        }
                        else
                        {
                                Mass = 0;
                        }
                }

    //Rotate the shape
    public override void Rotate(float angle, Vector2 originTranslation = new
Vector2())
    {
      Angle += angle;
      //Angles in radians given from 0 to 2 * PI, correct angles to within the
range
      if (Angle > 2 * (float)Math.PI)
        Angle -= 2 * (float)Math.PI;
      else if (Angle < 0)
        Angle += 2 * (float)Math.PI;
    }

                public float Radius {get; private set;}

    //Call to the SetAABB method
                protected override void CalculateAABB()
                {
                        SetAABB(Radius, Radius);
                }

                protected override void CalculateMass()
                {
      //Calculate the mass using density * volume (Since volume is area, we
multiply by a set depth)
                        Mass = Volume * Density * 0.0000002f;

      //If mass exceeds the limits then set them to the limit they exceed
                        if (Mass > Limits.MAXIMUM_MASS)
                                Mass = Limits.MAXIMUM_MASS;
                        else if (Mass < Limits.MINIMUM_MASS)
                                Mass = Limits.MINIMUM_MASS;
                }

    protected override void ComputeInertia()
    {
```

```
        //Circle inertia = 1/2 * (mass * radius^2)
        Inertia = 0.5f * Mass * (Radius * Radius);
    }

    protected override void CalculateCenterOfMass()
    {
    }

    //Calculate volume in the mathsUtility file
    protected override void CalculateVolume()
    {
        Volume = mathsUtility.CircleVolume(Radius);
    }
        }
}
```

The physics shape circle class is used to set up a circle's properties. It sets the shape to the integer corresponding to circle, sets the material to the parameter given as well as setting the radius and whether gravity affects it. It uses the material to set its coefficients for restitution (bounciness), friction (static and dynamic) and its density.

A Rotate method is included to set the angle based on a parameter given, it then corrects this to be within the range $0 \leq \text{angle} \leq 2\ \pi$ (angles are given in radians). It sets the axis-aligned bounding box to have a radius half-height and a radius half-width (AKA the height and width are the diameter of the circle). It calculates the mass using

*Mass = Density * Volume*

The float afterwards is an arbitrary scalar. It then calculate the inertia using

*Inertia = 0.5 * Mass * Radius*^{2}

It also calls the mathsUtility class to calculate the circle's volume.

PhysicsShapePolygon Code and Explanation

```
using System;
using Microsoft.Xna.Framework;

using NEA_Physics_Engine.Physics.Properties;
using NEA_Physics_Engine.Utilities;
using NEA_Physics_Engine.Rendering;
using System.Diagnostics;

namespace NEA_Physics_Engine.Physics
{
        public sealed class PhysicsPolygonDefine : PhysicsShape
        {
    //Attributes
    //my_Vertices: an array of vertices

    //Methods
    //PhysicsPolygonDefine: used to define properties of the polygon
    //Rotate: used to rotate the object
    //VerticesCount: get/set the vertex count of the polygon
    //Vertices: get an array of vertices
    //GetVertex: returns a given vertex
    //CalculateAABB: used to set data about the axis-aligned bounding box
    //CalculateMass: used to calculate the mass of the polygon
    //ComputeInertia: used to calculate the inertia of the polygon
    //CalculateCentreOfMass: used to calculate the centre of mass of the polygon
    //CalculateVolume: used to calculate the volume of the polygon via a
redirect to the mathsUtility

    //Get the properties of the polygon
                public PhysicsPolygonDefine(PhysicsPolygonDefine other)
                {
                        Shape = other.Shape;
                        Mass = other.Mass;
                        VerticesCount = other.VerticesCount;
                        my_Vertices = other.my_Vertices;
      Restitution = other.Restitution;
      StaticFriction = other.StaticFriction;
      DynamicFriction = other.DynamicFriction;
      Inertia = other.Inertia;
      Gravity = other.Gravity;
                }

                public PhysicsPolygonDefine(Vector2[] parameter_Vertices, bool
isStatic = false, int parameter_Material = Properties.Material.SOLID, bool
gravityAffects = true)
                {
      if (parameter_Vertices.Length < 3)
```

```
        {
            RenderManager.Instance.DrawString("Polygon must have at least 3
vertices");
        }

        //Set properties of the polygon that aren't given
                        Shape = 2;
                        Material = parameter_Material;

                        VerticesCount = parameter_Vertices.GetLength(0);
                        my_Vertices = parameter_Vertices;
        Gravity = gravityAffects;

        //Restitution and Friction affected by material, case not used as require
jump statements
        if (Material == 1)
        {
            Restitution = Properties.Restitution.SOLID;
            StaticFriction = Properties.StaticFriction.SOLID;
            DynamicFriction = Properties.DynamicFriction.SOLID;
            Density = Properties.Density.SOLID;
        }
        if (Material == 2)
        {
            Restitution = Properties.Restitution.ICE;
            StaticFriction = Properties.StaticFriction.ICE;
            DynamicFriction = Properties.DynamicFriction.ICE;
            Density = Properties.Density.ICE;
        }
        if (Material == 3)
        {
            Restitution = Properties.Restitution.WOOD;
            StaticFriction = Properties.StaticFriction.WOOD;
            DynamicFriction = Properties.DynamicFriction.WOOD;
            Density = Properties.Density.WOOD;
        }
        if (Material == 4)
        {
            Restitution = Properties.Restitution.METAL;
            StaticFriction = Properties.StaticFriction.METAL;
            DynamicFriction = Properties.DynamicFriction.METAL;
            Density = Properties.Density.METAL;
        }
        if (Material == 5)
        {
            Restitution = Properties.Restitution.RUBBER;
            StaticFriction = Properties.StaticFriction.RUBBER;
            DynamicFriction = Properties.DynamicFriction.RUBBER;
            Density = Properties.Density.RUBBER;
```

```
        }


        //Calculate the volume, the bounding box and the center of mass
                        CalculateVolume();
                        CalculateAABB();
                        CalculateCenterOfMass();

        //Calculate mass using density and volume, unless object is static (in
which case there is no mass, so no effect by gravity)
                        if (isStatic == false)
                        {
                                CalculateMass();
                        }
                        else
                        {
                                Mass = 0;
                        }
                }

                private Vector2[] my_Vertices;
                public int VerticesCount {get; private set;}
                public Vector2[] Vertices {get {return my_Vertices;}}
                public Vector2 GetVertex(int parameter_Index) {return
my_Vertices[parameter_Index];}

      //Rotate the shape
      public override void Rotate(float angle, Vector2 origin = new Vector2())
      {
        //Angles in radians given from 0 to 2 * PI, correct angles to within the
range
        Angle += angle;
        if (Angle > 2 * (float)Math.PI)
          Angle -= 2 * (float)Math.PI;
        else if (Angle < 0)
          Angle += 2 * (float)Math.PI;

        float tempX;
        for (int i = 0; i < VerticesCount; i++)
        {
          tempX = my_Vertices[i].X;
          my_Vertices[i].X = (float)(Math.Cos(-angle) * tempX - Math.Sin(-angle) *
my_Vertices[i].Y);
          my_Vertices[i].Y = (float)(Math.Sin(-angle) * tempX + Math.Cos(-angle) *
my_Vertices[i].Y);
        }
      }

                protected override void CalculateAABB()
```

```
            {
        //Initialise the minimum and maximum points to 0
                        float minimumX = 0;
                        float minimumY = 0;
                        float maximumX = 0;
                        float maximumY = 0;

        //For every vertex of the polygon, check if the points X or Y co-ordinate is
a new minimum or maximum then set new min/max
                        for (int i = 0; i < VerticesCount; i++)
                        {
                                if (my_Vertices[i].X < minimumX)
                                        minimumX = my_Vertices[i].X;

                                if (my_Vertices[i].Y < minimumY)
                                        minimumY = my_Vertices[i].Y;

                                if (my_Vertices[i].X > maximumX)
                                        maximumX = my_Vertices[i].X;

                                if (my_Vertices[i].Y > maximumY)
                                        maximumY = my_Vertices[i].Y;
                        }

        //Set the bounding box
                        SetAABB(minimumX + maximumX, minimumY +
maximumY);
                }

                protected override void CalculateMass()
                {
        //Calculate the mass using density * volume (Since volume is area, we
multiply by a set depth)
        Mass = Volume * Density * 0.0000002f;

        //If mass exceeds the limits then set them to the limit they exceed
                        if (Mass > Limits.MAXIMUM_MASS)
                                Mass = Limits.MAXIMUM_MASS;
                        else if (Mass < Limits.MINIMUM_MASS)
                                Mass = Limits.MINIMUM_MASS;
                }

                protected override void CalculateCenterOfMass()
                {
        //Use the polygon center method in the mathsUtility to find the center
                        Vector2 center = mathsUtility.PolygonCenter(Vertices,
Volume);

        //Subtract center from the vertices
```

```
                    for (int i = 0; i < VerticesCount; i++)
                        my_Vertices[i] -= center;
                }

    protected override void ComputeInertia()
    {
      Inertia = mathsUtility.PolygonInertia(Vertices, Mass);
      if (Inertia > Limits.MAXIMUM_INERTIA)
        Inertia = Limits.MAXIMUM_INERTIA;
      else if (Inertia < Limits.MINIMUM_INERTIA)
        Inertia = Limits.MINIMUM_INERTIA;
    }

    //Calculate volume in the mathsUtility file
            protected override void CalculateVolume()
            {
                Volume = mathsUtility.PolygonVolume(Vertices);
            }
      }
}
```

The physics shape polygon class is used to set up a polygon's properties. It sets the shape to the integer corresponding to polygon, sets the material to the parameter given as well as setting whether gravity affects it. It uses the material to set its coefficients for restitution (bounciness), friction (static and dynamic) and its density.

A Rotate method is included to set the angle based on a parameter given, it then corrects this to be within the range $0 \leq \text{angle} \leq 2\ \pi$ (angles are given in radians) and each point is set using the 2x2 matrix for a rotation. It then finds maximum and minimum points in order to set the axis-aligned bounding box. It sets the mass using

*Mass = Density * Volume*

The float afterwards is an arbitrary scalar. It then finds the center of mass and the inertia by calling the mathsUtility class.

PhysicsProperties Code and Explanation

```
namespace NEA_Physics_Engine.Physics.Properties
{
  //The possible types of each property of an object

        public static class Shape
        {
    //Used as a look-up for shape type
                public const int UNSPECIFIED = 0;
                public const int CIRCLE = 1;
                public const int POLYGON = 2;
                public const int NR_SHAPES = 3;
        }

        public static class Material
        {
    //Used as a look-up for material
                public const int UNSPECIFIED = 0;
                public const int SOLID = 1;
    public const int ICE = 2;
    public const int WOOD = 3;
    public const int METAL = 4;
    public const int RUBBER = 5;
                public const int NR_MATERIALS = 6;
        }

        public static class Density
        {
    //Used as a look-up for density
                public const int UNSPECIFIED = 0;
                public const int AIR = 2;
                public const int SOLID = 500;
    public const int ICE = 100;
    public const int WOOD = 700;
    public const int METAL = 2000;
    public const int RUBBER = 200;
        }

  public static class Restitution
  {
    //Used as a look-up for restitution
    public const float UNSPECIFIED = 0f;
    public const float AIR = 0f;
    public const float SOLID = 0.7f;
    public const float ICE = 0.3f;
    public const float WOOD = 0.2f;
    public const float METAL = 0.1f;
```

```
    public const float RUBBER = 0.8f;
  }

  public static class StaticFriction
  {
    //Used as a look-up for static friction
    public const float UNSPECIFIED = 0f;
    public const float AIR = 0f;
    public const float SOLID = 0.03f;
    public const float ICE = 0.001f;
    public const float WOOD = 0.05f;
    public const float METAL = 0.03f;
    public const float RUBBER = 0.05f;
  }

  public static class DynamicFriction
  {
    //Used as a look-up for dynamic friction
    public const float UNSPECIFIED = 0f;
    public const float AIR = 0f;
    public const float SOLID = 0.01f;
    public const float ICE = 0.0005f;
    public const float WOOD = 0.03f;
    public const float METAL = 0.02f;
    public const float RUBBER = 0.04f;
  }

      public static class Limits
      {
    //Used as a look-up for limits on different properties
            public const float MAXIMUM_FORCE = 1000.0f;
            public const float MINIMUM_FORCE = 0.0f;
            public const float MAXIMUM_LINEAR_VELOCITY = 500.0f;
            public const float MINIMUM_LINEAR_VELOCITY = 0.0f;
    public const float MAXIMUM_ANGULAR_VELOCITY = 5f;
    public const float MINIMUM_ANGULAR_VELOCITY = 1f;
            public const float MAXIMUM_MASS = 100.0f;
            public const float MINIMUM_MASS = 0.001f;
    public const int MAXIMUM_INERTIA = 900000000;
    public const int MINIMUM_INERTIA = 0;
      }

      public static class Units
      {
    //Set up for scale
            public const int PIXELS_PER_METER = 32;
            public const float PIXELS_PER_METER_INVERSE = (float)1 /
(float)PIXELS_PER_METER;
      }
```

}

This class is static so is unaffected throughout runtime. The main purpose is to be used as a look-up table for different properties. It sets each type of shape to an integer, as well as the materials. It then contains the density, restitution and frictional coefficients for each material. This is helpful as it means, when adding new materials or editing the properties of existing ones, instead of manually setting properties at instantiation of the object, you can add or edit the properties in one place.

It also contains the maximum and minimum limits for different physical properties such as velocity, force, etc. Again this allows limits to be edited in one place as opposed to multiple locations within the code.

PhysicsManager Code and Explanation

```
using System;
using System.Diagnostics;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Input;

using NEA_Physics_Engine.Physics.Properties;
using NEA_Physics_Engine.Input;
using NEA_Physics_Engine.Rendering;
using NEA_Physics_Engine.Utilities;
using NEA_Physics_Engine.Physics;

namespace NEA_Physics_Engine.Physics
{
   class PhysicsManager : Interfaces.Updateable
   {
      //Attributes
      //my_CollisionListeners (CollisionInfoDelegate): objects checking for
collisions
      //my_CollisionFunctionParameters (IsCollidingDelegate): objects to be
passed to the collision functions
      //my_ResolveFunctionParameters (CollisionResolveDelegate): objects to be
passed to the collision resolution function
      //my_Bodies (List of PhysicsBody): list of physics bodies in the scene
      //my_Gravity (Vector2): the gravity vector
      //my_GravityNormal (Vector2): vector perpendicular to gravity
      //bodySelected (PhysicsBody): object currently selected (used for testing
purposes)
      //showDebug (Boolean): boolean used to determine whether to display
debug information
      //lastMousePosition (Vector2): the position of the mouse on the last update
      //my_InputManager (InputManager): an instance of the input manager

      //Methods
      //RegisterListener: add an element to my_CollisionListeners
      //UnregisterListener: remove an element from my_CollisionListeners
      //Update: run on update to check for collisions and resolve them
      //AddBody: add a physics body to my_Bodies
      //Clear: clear my_Bodies
      //isColliding: check whether two objects are colliding
      //isCollidingCircleCircle: check whether two circles are colliding
      //isCollidingConvexConvex: check whether two polygons are colliding
      //isCollidingConvexCircle: check whether a polygon and a circle are colliding
      //isCollidingCircleConvex: check whether a circle and a polygon are colliding
      //ResolveSolid: resolves collisions between two solid objects
      //Gravity: get/set my_Gravity
```

```
//Projection structure used for tracking possible collisions
private struct Projection
{
    public Projection(float minimum, float maximum)
    {
        Minimum = minimum;
        Maximum = maximum;
    }
    public float Minimum;
    public float Maximum;
    public bool Overlap(Projection other, out float amount)
    {
        if (Maximum >= other.Minimum && other.Maximum >= Minimum)
        {
            if (Maximum < other.Maximum)
                amount = Math.Abs(Maximum - other.Minimum);
            else
                amount = Math.Abs(other.Maximum - Minimum);
            return true;
        }
        else
        {
            amount = -1;
            return false;
        }
    }
}

private Projection Project(Vector2 position, Vector2[] vertices, Vector2 axis)
{
    float minimum = Vector2.Dot(position + vertices[0], axis);
    float maximum = minimum;
    //Find maximum and minimum
    for (int i = 1; i < vertices.Length; i++)
    {
        // NOTE: the axis must be normalized to get accurate projections
        float p = Vector2.Dot(position + vertices[i], axis);
        if (p < minimum)
            minimum = p;
        else if (p > maximum)
            maximum = p;
    }
    return new Projection(minimum, maximum);
}

//Structure for storing details about the collision
public struct CollisionInfo
{
    public Vector2 CollisionNormal;
```

```
        public Vector2 Overlapping;
        public List<Vector2> CollisionPoints;
        public List<Vector2> IntersectionPoints;
    }

    private delegate bool IsCollidingDelegate(PhysicsBody bodyA, PhysicsBody
bodyB, out CollisionInfo collisionInfo);
    private delegate void CollisionResolveDelegate(PhysicsBody bodyA,
PhysicsBody bodyB, CollisionInfo collisionInfo);
    public delegate void CollisionInfoDelegate(CollisionInfo collisionInfo,
PhysicsBody bodyA, PhysicsBody bodyB);
    private CollisionInfoDelegate my_CollisionListeners = null;
    private IsCollidingDelegate[,] my_CollisionFunctionParameters;
    private CollisionResolveDelegate[] my_ResolveFunctionParameters;
    //Store a list of objects
    private List<PhysicsBody> my_Bodies;

    //Set gravitational constant and gravitational normal
    private static Vector2 my_Gravity = new Vector2(0, 20.0f);
    private static Vector2 my_GravityNormal =
mathsUtility.leftPerpendicular(my_Gravity);

    PhysicsBody bodySelected = null;
    private bool showDebug = false;
    Vector2 lastMousePosition;
    InputManager my_InputManager = InputManager.Instance;

    public void RegisterListener(CollisionInfoDelegate parameter_Listener)
    {
        my_CollisionListeners += parameter_Listener;
    }
    public void UnregisterListener(CollisionInfoDelegate parameter_Listener)
    {
        my_CollisionListeners -= parameter_Listener;
    }


    public virtual void Update()
    {
        CollisionInfo collisionInfo;

        //Get mouse position from an instance of the input manager
        Vector2 mousePosition = my_InputManager.GetMousePosition();

        //Check for gravity manipulation inputs
        if (my_InputManager.KeyIsPressed(Keys.W))
        {
            my_Gravity = new Vector2(my_Gravity.X, my_Gravity.Y - 0.25f);
        }
```

```
        if (my_InputManager.KeyIsPressed(Keys.S))
        {
            my_Gravity = new Vector2(my_Gravity.X, my_Gravity.Y + 0.25f);
        }
        if (my_InputManager.KeyIsPressed(Keys.A))
        {
            my_Gravity = new Vector2(my_Gravity.X - 0.25f, my_Gravity.Y);
        }
        if (my_InputManager.KeyIsPressed(Keys.D))
        {
            my_Gravity = new Vector2(my_Gravity.X + 0.25f, my_Gravity.Y);
        }
        if (my_InputManager.KeyIsPressed(Keys.Enter))
        {
            my_Gravity = new Vector2(0, 20.0f);
        }


        if (showDebug == true)
        {
            //Draw debug strings
            if (bodySelected != null)
            {
                RenderManager.Instance.DrawString("   Position: " +
bodySelected.Position);
                RenderManager.Instance.DrawString("   Linear Velocity: " +
bodySelected.linearVelocity);
                RenderManager.Instance.DrawString("   Mass: " + bodySelected.Mass);
                RenderManager.Instance.DrawString("   Restitution: " +
bodySelected.Restitution);
                RenderManager.Instance.DrawString("   Static Friction: " +
bodySelected.StaticFriction);
                RenderManager.Instance.DrawString("   Dynamic Friction: " +
bodySelected.DynamicFriction);
                RenderManager.Instance.DrawString("   Angular Velocity: " +
bodySelected.angularVelocity);
                RenderManager.Instance.DrawString("   Material: " +
bodySelected.ShapeDefine.Material);
                RenderManager.Instance.DrawString("   Volume: " +
bodySelected.ShapeDefine.Volume);
            }
            RenderManager.Instance.DrawString("   Gravity: " + my_Gravity);
        }

        if (bodySelected != null)
        {
            if (InputManager.Instance.KeyIsPressed(Keys.Right))
                bodySelected.AddForce(new Vector2(50 * bodySelected.Mass, 0),
bodySelected.Position, FORCE_TYPE.IMPULSE);
```

```
            if (InputManager.Instance.KeyIsPressed(Keys.Left))
               bodySelected.AddForce(new Vector2(-50 * bodySelected.Mass, 0),
bodySelected.Position, FORCE_TYPE.IMPULSE);
            if (InputManager.Instance.KeyIsPressed(Keys.Up))
               bodySelected.AddForce(new Vector2(0, -50 * bodySelected.Mass),
bodySelected.Position, FORCE_TYPE.IMPULSE);
            if (InputManager.Instance.KeyIsPressed(Keys.Down))
               bodySelected.AddForce(new Vector2(0, 50 * bodySelected.Mass),
bodySelected.Position, FORCE_TYPE.IMPULSE);
         }


         //Debug toggle
         if (my_InputManager.KeyWasPressed(Keys.M))
         {
            if (showDebug == false)
               showDebug = true;
            else
               showDebug = false;
         }


         //Add forces for each object
         foreach (PhysicsBody bodyA in my_Bodies)
         {
            if (bodyA.isStatic() == false)
            {
               foreach (PhysicsBody bodyB in my_Bodies)
               {
                  //Check if bodies are colliding and resolve if neccesary
                  if (bodyA.GetID() != bodyB.GetID() && isColliding(bodyA, bodyB, out
collisionInfo))
                  {
                     my_ResolveFunctionParameters[bodyB.Material].Invoke(bodyA,
bodyB, collisionInfo);
                  }
               }

               //Select object for debug
               if (my_InputManager.LeftMouseIsPressed() &&
Math.Abs(mousePosition.X - bodyA.Position.X) < 10
                           && Math.Abs(mousePosition.Y - bodyA.Position.Y) < 10)
               {
                  bodySelected = bodyA;
               }

               if (bodyA.ShapeDefine.Gravity == true)
               {
```

```
                if (bodyA.Shape == 2)
                {
                    Vector2 CentreOfGravity = new Vector2();
                    for (int i = 0; i < bodyA.PolygonDefine.VerticesCount; i++)
                    {
                        CentreOfGravity += bodyA.PolygonDefine.Vertices[i];
                    }
                    CentreOfGravity *= 1 / bodyA.PolygonDefine.VerticesCount;
                    bodyA.AddForce((my_Gravity) * bodyA.Mass, bodyA.Position,
FORCE_TYPE.IMPULSE);
                }
                else
                {
                    //Add gravity
                    bodyA.AddForce((my_Gravity) * bodyA.Mass, bodyA.Position,
FORCE_TYPE.IMPULSE);
                }
            }


            //Update properties of body
            bodyA.Update();
        }
        //By next update, current mouse position will be previous mouse
position
        lastMousePosition = mousePosition;
      }
    }

    public void AddBody(PhysicsBody phys_body)
    {
      //Add a new object to the objects list
      my_Bodies.Add(phys_body);
    }

    public void Clear()
    {
      //Clear the objects list
      my_Bodies.Clear();
    }

    //----------------*******COLLISION*******----------------
    private bool isColliding(PhysicsBody bodyA, PhysicsBody bodyB, out
CollisionInfo collisionInfo)
    {
      return my_CollisionFunctionParameters[bodyA.Shape,
bodyB.Shape].Invoke(bodyA, bodyB, out collisionInfo);
    }
```

```
    private bool isCollidingCircleCircle(PhysicsBody bodyA, PhysicsBody bodyB,
out CollisionInfo collisionInfo)
    {
      collisionInfo.CollisionPoints = new List<Vector2>();
      collisionInfo.IntersectionPoints = new List<Vector2>();

      //Get distance between centres of circles
      Vector2 distance = bodyA.Position - bodyB.Position;
      //Get sum of radii
      float widths = bodyA.CircleDefine.Radius + bodyB.CircleDefine.Radius;

      //if the distance between centres is less than the width then they are
intersecting
      if (distance.LengthSquared() < widths * widths)
      {
        if (distance.X == 0 && distance.Y == 0)
        {
          //Since the circles have equal positions a "Special distance" is
required to create a collision normal
          Vector2 specialDistance = bodyA.LastPosition - bodyB.Position;
          if (specialDistance.X == 0 && specialDistance.Y == 0)
            specialDistance.X = 1;
          else
            specialDistance.Normalize();

          collisionInfo.Overlapping = specialDistance * widths;
          collisionInfo.CollisionNormal = specialDistance;
          collisionInfo.CollisionPoints = new List<Vector2>();
        }
        else
        {
          //Check how much the circles are intersecting and set collision info
accordingly
          float overlappingAmount = widths - distance.Length();
          distance.Normalize();
          collisionInfo.Overlapping = distance * overlappingAmount;
          collisionInfo.CollisionNormal = distance;
          collisionInfo.CollisionPoints.Add(bodyB.Position +
(collisionInfo.CollisionNormal * bodyB.CircleDefine.Radius));
        }
        return true;
      }
      else
      {
        collisionInfo.CollisionNormal = new Vector2();
        collisionInfo.Overlapping = new Vector2();
        return false;
      }
    }
```

```
    private bool isCollidingConvexConvex(PhysicsBody bodyA, PhysicsBody
bodyB, out CollisionInfo collisionInfo)
    {
        collisionInfo.CollisionNormal = new Vector2();
        collisionInfo.Overlapping = new Vector2();
        collisionInfo.CollisionPoints = new List<Vector2>();
        collisionInfo.IntersectionPoints = new List<Vector2>();

        List<Vector2> verticesInside = new List<Vector2>();

        Vector2 edge;
        Vector2 edgeNormal;

        Vector2 shortestOverlapAxis = bodyA.PolygonDefine.GetVertex(0);

        Projection projectionA;
        Projection projectionB;
        float shortestOverlapAmount = int.MaxValue;
        float overlapAmount = 0;

        //Check every vertex in the first body
        for (int i = 0; i < bodyA.PolygonDefine.VerticesCount; i++)
        {
            //Get the vector the side that is being checked
            edge = bodyA.PolygonDefine.GetVertex((i + 1 ==
bodyA.PolygonDefine.VerticesCount ? 0 : i + 1)) -
bodyA.PolygonDefine.GetVertex(i);
            //Get its normal direction
            edgeNormal = mathsUtility.rightPerpendicular(edge);
            edgeNormal.Normalize();

            //Create projections
            projectionA = Project(bodyA.Position, bodyA.PolygonDefine.Vertices,
edgeNormal);
            projectionB = Project(bodyB.Position, bodyB.PolygonDefine.Vertices,
edgeNormal);

            //If projections don't overlap then not colliding
            if (!projectionA.Overlap(projectionB, out overlapAmount))
                return false;
            else if (overlapAmount < shortestOverlapAmount)
            {
                //Make sure the shortest overlap is found
                shortestOverlapAxis = edgeNormal;
                shortestOverlapAmount = overlapAmount;
            }

            //Check side against every vertex in B
```

```
            for (int j = 0; j < bodyB.PolygonDefine.VerticesCount; j++)
            {
                //Represent the 2 lines as 4 points
                Vector2 point1 = (bodyB.PolygonDefine.GetVertex(j) +
bodyB.Position);
                Vector2 point2 = (bodyB.PolygonDefine.GetVertex((j + 1 ==
bodyB.PolygonDefine.VerticesCount ? 0 : j + 1)) + bodyB.Position);
                Vector2 point3 = (bodyA.PolygonDefine.GetVertex(i) +
bodyA.Position);
                Vector2 point4 = (bodyA.PolygonDefine.GetVertex((i + 1 ==
bodyA.PolygonDefine.VerticesCount ? 0 : i + 1)) + bodyA.Position);
                Vector2 intersectionPoint;

                //Find if the sides are intersecting and if so add the intersection point
to the list
                if (mathsUtility.LineLineIntersection(point1, point2, point3, point4,
out intersectionPoint))
                    collisionInfo.IntersectionPoints.Add(intersectionPoint);
            }

            //Find the vertices of B which are inside A
            if (i == 0)
            {
                for (int j = 0; j < bodyB.PolygonDefine.VerticesCount; j++)
                {
                    Vector2 vertex = (bodyB.PolygonDefine.GetVertex(j) +
bodyB.Position) - (bodyA.PolygonDefine.GetVertex(i) + bodyA.Position);
                    float dot = Vector2.Dot(vertex, edgeNormal);

                    if (dot < 0)
                        verticesInside.Add(bodyB.PolygonDefine.GetVertex(j) +
bodyB.Position);
                }
            }
            else
            {
                for (int j = 0; j < verticesInside.Count; j++)
                {
                    Vector2 vertex = verticesInside[j] -
(bodyA.PolygonDefine.GetVertex(i) + bodyA.Position);
                    float dot = Vector2.Dot(vertex, edgeNormal);

                    if (dot > 0)
                    {
                        verticesInside.RemoveAt(j);
                        j--;
                    }
                }
            }
```

```
        }

        for (int i = 0; i < bodyB.PolygonDefine.VerticesCount; i++)
        {
            //Calculate the Vector of B's side and find the normal
            edge = bodyB.PolygonDefine.GetVertex((i + 1 ==
bodyB.PolygonDefine.VerticesCount ? 0 : i + 1)) -
bodyB.PolygonDefine.GetVertex(i);
            edgeNormal = mathsUtility.rightPerpendicular(edge);
            edgeNormal.Normalize();

            //Just swap
            projectionB = Project(bodyA.Position, bodyA.PolygonDefine.Vertices,
edgeNormal);
            projectionA = Project(bodyB.Position, bodyB.PolygonDefine.Vertices,
edgeNormal);

            if (!projectionA.Overlap(projectionB, out overlapAmount))
                return false;
            else if (overlapAmount < shortestOverlapAmount)
            {
                shortestOverlapAxis = edgeNormal;
                shortestOverlapAmount = overlapAmount;
            }

            if (i == 0)
            {
                for (int j = 0; j < bodyA.PolygonDefine.VerticesCount; j++)
                {
                    Vector2 vertex = (bodyA.PolygonDefine.GetVertex(j) +
bodyA.Position) - (bodyB.PolygonDefine.GetVertex(i) + bodyB.Position);
                    float dot = Vector2.Dot(vertex, edgeNormal);

                    if (dot < 0)
                        verticesInside.Add(bodyA.PolygonDefine.GetVertex(j) +
bodyA.Position);
                }
            }
            else
            {
                for (int j = 0; j < verticesInside.Count; j++)
                {
                    Vector2 vertex = verticesInside[j] -
(bodyB.PolygonDefine.GetVertex(i) + bodyB.Position);
                    float dot = Vector2.Dot(vertex, edgeNormal);
                    if (dot > 0)
                    {
                        verticesInside.RemoveAt(j);
                        j--;
```

```
                }
            }
        }
    }

    //Set collision info
    collisionInfo.CollisionNormal = shortestOverlapAxis;
    collisionInfo.Overlapping = shortestOverlapAxis * shortestOverlapAmount;
    collisionInfo.CollisionPoints.AddRange(verticesInside);

    return true;
}

private bool isCollidingConvexCircle(PhysicsBody polygon, PhysicsBody
circle, out CollisionInfo collisionInfo)
{
    collisionInfo.CollisionPoints = new List<Vector2>();
    collisionInfo.IntersectionPoints = new List<Vector2>();

    //Used when circle inside polygon
    Vector2 closestProjection = new
Vector2(ProgramManager.Instance.ScreenWidth,
ProgramManager.Instance.ScreenHeight);
    bool insidePolygon = true;
    for (int i = 0; i < polygon.PolygonDefine.VerticesCount; i++)
    {
        //Vector from current edge to center of circle
        Vector2 vectorToCircle = circle.Position - (polygon.Position +
polygon.PolygonDefine.GetVertex(i));
        Vector2 currentEdge = polygon.PolygonDefine.GetVertex(i + 1 ==
polygon.PolygonDefine.VerticesCount ? 0 : i + 1) -
polygon.PolygonDefine.GetVertex(i);
        //Length of current edge squared
        float currentEdgeLengthSquared = currentEdge.LengthSquared();
        currentEdge.Normalize();
        float circleToEdgeProjection = Vector2.Dot(vectorToCircle,
currentEdge);

        if (circleToEdgeProjection > 0)
        {
            if ((circleToEdgeProjection * circleToEdgeProjection) <
currentEdgeLengthSquared)
            {
                Vector2 vectorCircleProjectionEdge = currentEdge *
circleToEdgeProjection;
                Vector2 projectionToCircle = vectorToCircle -
vectorCircleProjectionEdge;
                float projectionToCircleLengthSquared =
projectionToCircle.LengthSquared();
```

```
                if (projectionToCircleLengthSquared <
closestProjection.LengthSquared())
                    closestProjection = projectionToCircle;

                if (projectionToCircleLengthSquared < circle.CircleDefine.Radius *
circle.CircleDefine.Radius)
                {
                    collisionInfo.CollisionNormal = - projectionToCircle;
                    collisionInfo.CollisionNormal.Normalize();
                    collisionInfo.CollisionPoints.Add(circle.Position +
(collisionInfo.CollisionNormal * circle.CircleDefine.Radius));
                    collisionInfo.Overlapping = collisionInfo.CollisionNormal *
(circle.CircleDefine.Radius - projectionToCircle.Length());
                    return true;
                }
            }
        }
        else if (vectorToCircle.LengthSquared() < circle.CircleDefine.Radius *
circle.CircleDefine.Radius)
        {
            collisionInfo.CollisionNormal = - vectorToCircle;
            collisionInfo.CollisionNormal.Normalize();
            collisionInfo.CollisionPoints.Add(polygon.PolygonDefine.GetVertex(i)
+ polygon.Position);
            collisionInfo.Overlapping = collisionInfo.CollisionNormal *
(circle.CircleDefine.Radius - vectorToCircle.Length());
            return true;
        }
        Vector2 edgeNormal = mathsUtility.rightPerpendicular(currentEdge);

        if (Vector2.Dot(vectorToCircle, edgeNormal) > 0)
            insidePolygon = false;
        else if (i == polygon.PolygonDefine.VerticesCount - 1)
        {
            collisionInfo.CollisionNormal = vectorToCircle;
            collisionInfo.CollisionNormal.Normalize();
            collisionInfo.Overlapping = vectorToCircle * (vectorToCircle.Length()
+ circle.CircleDefine.Radius);
            collisionInfo.CollisionPoints.Add(polygon.PolygonDefine.GetVertex(0)
+ polygon.Position);
            return insidePolygon;
        }
    }
    collisionInfo.CollisionNormal = new Vector2();
    collisionInfo.Overlapping = new Vector2();
    return false;
}
```

```
    private bool isCollidingCircleConvex(PhysicsBody bodyA, PhysicsBody
bodyB, out CollisionInfo collisionInfo)
    {
        //Dummy, redirect to isCollidingConvexCircle-function.
        return isCollidingConvexCircle(bodyB, bodyA, out collisionInfo);
    }

    private void ResolveSolid(PhysicsBody bodyA, PhysicsBody bodyB,
CollisionInfo collisionInfo)
    {
        //Make sure normal is in right direction
        if (Vector2.Dot(bodyA.Position - bodyB.Position,
collisionInfo.CollisionNormal) < 0)
        {
            collisionInfo.CollisionNormal = -collisionInfo.CollisionNormal;
            collisionInfo.Overlapping = -collisionInfo.Overlapping;
        }
        //Translate minimum distance so not overlapping
        bodyA.Position += collisionInfo.Overlapping;

        float j = 0;
        float jt = 0;
        float CoefficientOfFriction;
        Vector2 collisionPointRelativeVelocityAtoB = Vector2.UnitX;
        Vector2 collisionTangent = new Vector2();
        //Not 100% accurate if lands on flat surface because collision-points are
calculated one at a time, would be true even if done via a queue
        foreach (Vector2 collisionPoint in collisionInfo.CollisionPoints)
        {
            //First get distance-vector from each body-center to collision-point
            Vector2 collisionPointRadiusA = collisionPoint - bodyA.Position;
            Vector2 collisionPointRadiusB = collisionPoint - bodyB.Position;

            //Calculate vector perpendicular to the vector from mass-center to
collision-point
            Vector2 collisionPointRadiusAPerpendicular =
mathsUtility.rightPerpendicular(collisionPointRadiusA);
            Vector2 collisionPointRadiusBPerpendicular =
mathsUtility.rightPerpendicular(collisionPointRadiusB);

            //Velocity at collisionpoint
            Vector2 collisionPointVelocityA = bodyA.linearVelocity;
            Vector2 collisionPointVelocityB = bodyB.linearVelocity;

            //Relative velocity of bodyA's with respect to bodyB's collision-point
velocity
            collisionPointRelativeVelocityAtoB = collisionPointVelocityA -
collisionPointVelocityB;
```

```
            //Find a tangent to the collision (for frictional purposes)
            collisionTangent = collisionPointRelativeVelocityAtoB -
Vector2.Dot(collisionPointRelativeVelocityAtoB, collisionInfo.CollisionNormal) *
collisionInfo.CollisionNormal;
            collisionTangent.Normalize();


            //Calculate new impulse
            j = Vector2.Dot(-(1 + (bodyA.Restitution + bodyB.Restitution) * 0.5f) *
collisionPointRelativeVelocityAtoB, collisionInfo.CollisionNormal)
                    / (Vector2.Dot(collisionInfo.CollisionNormal,
(collisionInfo.CollisionNormal * (bodyA.InverseMass + bodyB.InverseMass))));

            //Calculate tangential force
            jt = -(1 + (bodyA.Restitution + bodyB.Restitution) * 0.5f ) *
Vector2.Dot(collisionPointRelativeVelocityAtoB,
collisionTangent)/(bodyA.InverseMass + bodyB.InverseMass);

            //Calculate coefficient of friction for the collision
            CoefficientOfFriction = (bodyA.StaticFriction + bodyB.StaticFriction) *
0.5f;

            //Check if tangential force is great enough to overcome friction
            if (Math.Abs(jt) >= j * CoefficientOfFriction)
            {
               //Modify coefficient of friction to reflect dynamic friction
               CoefficientOfFriction = (bodyA.DynamicFriction +
bodyB.DynamicFriction) * 0.5f;
               jt = -j * CoefficientOfFriction;
            }

            //"Temp"-fix for j (make sure sign is correct)
            if (Vector2.Dot(j * collisionInfo.CollisionNormal,
collisionInfo.CollisionNormal) < 0)
               j = -j;

            //Apply force (torque manually calculated by the body itself)
            bodyA.AddForce(j * collisionInfo.CollisionNormal, collisionPoint,
FORCE_TYPE.IMPULSE);

            if (Math.Abs(jt) < Limits.MAXIMUM_FORCE)
               bodyA.AddForce(jt * collisionTangent, collisionPoint,
FORCE_TYPE.IMPULSE);

            bodyB.AddForce(-j * collisionInfo.CollisionNormal, collisionPoint,
FORCE_TYPE.IMPULSE);
         }
      }
```

```
        public Vector2 Gravity
        {
            get { return my_Gravity; }
            set { my_Gravity = value; }
        }

        //Set up instance
        private static volatile PhysicsManager my_Instance;
        private static object my_SyncRoot = new Object();
        private PhysicsManager()
        {
            my_Bodies = new List<PhysicsBody>();

            //Instantiate matrix of function pointers to different collision-functions
            my_CollisionFunctionParameters = new
IsCollidingDelegate[Shape.NR_SHAPES, Shape.NR_SHAPES];
            //If two circles collide, use "CircleCircle" function
            my_CollisionFunctionParameters[Shape.CIRCLE, Shape.CIRCLE] = new
IsCollidingDelegate(isCollidingCircleCircle);
            //If two convex shapes collide, use "ConvexConvex" function
            my_CollisionFunctionParameters[Shape.POLYGON, Shape.POLYGON] =
new IsCollidingDelegate(isCollidingConvexConvex);
            //If two convex and circle shapes collide, use "ConvexCircle" function
            my_CollisionFunctionParameters[Shape.POLYGON, Shape.CIRCLE] = new
IsCollidingDelegate(isCollidingConvexCircle);
            //If two circle and convex shapes collide, use "CircleConvex" function
(Redirects to ConvexCircle)
            my_CollisionFunctionParameters[Shape.CIRCLE, Shape.POLYGON] = new
IsCollidingDelegate(isCollidingCircleConvex);

            my_ResolveFunctionParameters = new
CollisionResolveDelegate[Properties.Material.NR_MATERIALS];

            //Need to set for each possible material, set for being resolved as solids,
could expand for interactions with liquids or non-rigid bodies
            my_ResolveFunctionParameters[Material.SOLID] = new
CollisionResolveDelegate(ResolveSolid);
            my_ResolveFunctionParameters[Material.ICE] = new
CollisionResolveDelegate(ResolveSolid);
            my_ResolveFunctionParameters[Material.WOOD] = new
CollisionResolveDelegate(ResolveSolid);
            my_ResolveFunctionParameters[Material.METAL] = new
CollisionResolveDelegate(ResolveSolid);
            my_ResolveFunctionParameters[Material.RUBBER] = new
CollisionResolveDelegate(ResolveSolid);

            my_GravityNormal.Normalize();
        }
        public static PhysicsManager Instance
```

```
    {
      get
      {
        if (my_Instance == null)
          lock (my_SyncRoot)
            if (my_Instance == null)
            {
              my_Instance = new PhysicsManager();
            }
        return my_Instance;
      }
    }
  }
}
```

This is likely the most important class in the entire project as it takes care of collision detection and resolution as well as applying gravity and friction. The first things to talk about are the two custom structures that were written, the Projection and the CollisionInfo (highlighted in yellow). The projection contains a minimum and maximum point and these are used to project the movement of an object. This is used in what is known as a broad-phase check. The collision info contains a vector that represents the collision normal, a vector that represents the overlap between two objects and two lists of vectors (one for colliding points and one for intersecting points).

Another thing to mention is how the delegate type works. It represents a method that takes a specific set of parameters. This allows methods of that type to be passed as parameters and due to this, they are often used for handling events. Here they are used for collision and resolution events. (delegate type variables are highlighted in green)

The first two methods are for registering and unregistering collision listeners, this requires one of the delegate types to be passed. The next method is the update method. It starts by checking for any changes to the gravity vector and checking whether the debug info should be displayed. Next it checks if the user is adding any force to objects. It then checks for collisions, if the shape is a polygon then it finds the centre of gravity and if the shape is affected by gravity then a force is applied to the object based on the gravity vector. It then calls each bodies update function.

The isCollidingCircleCircle (highlighted in blue) method is called when two circles are passed to the isColliding function. It gets the distance between the centres of the two circles and the sum of the two radii (i.e. the minimum distance they can be from one another without colliding). It then checks if the distance between the centres squared is less than the sum of the radii squared (Squared is used to avoid taking square roots), if it is then they are colliding. A check is then used to check if the distance between the centres is zero, if it is then a special distance needs to be used. The special distance is the distance between the two

circles during the previous update but if the distance there was also zero then the special distance is set to (1, 0). The vector for the special distance is normalised. The overlap vector is then set to the special distance multiplied by the sum of the radii. The collision normal is the special distance vector.

If the distance wasn't zero then the collision info properties are set accordingly instead of being based on a special distance. If the distance is greater than the sum of the radii then the circles aren't colliding and a false is returned.

The isCollidingConvexConvex method (highlighted in red) is called when two polygons are passed to the isColliding method. Two projections are created, one for each polygon. A broad-phase check is then carried out (i.e. the projections are compared to see if there is any overlap) if the projections don't overlap then the two polygons aren't compared further and a false is returned. Otherwise a narrow-phase check is carried out on the polygons. The edges of the second body are then compared to the edges of the first body for any intersections and adds these intersections to the collision info. Another check is used to find the points of the second body inside the first. Finally the two roles are reversed to find all intersection points and all vertices that encroach on the other body's space.

The isCollidingConvexCircle method (highlighted in pink) is called when a polygon and a circle are passed to the isColliding method. The method draws a vector from the first vertex to the centre of the circle. It then creates an edge using the first two vertices and gets the length squared (again used to avoid the square root). Next it creates a circle to edge projection and checks if its greater than zero. It then creates a vector from the projection to the circle and checks its length and if its shorter than the shortest projection then the shortest projection is set to this vector. If the vector from the projection to the circle is shorter than the radius of the circle then the shapes are colliding. Otherwise if the vector to the circle is shorter than the radius then the shapes are colliding. Otherwise a normal vector to the edge is found and dotted with the vector to the circle. If this dot product is greater than zero then the shapes aren't colliding, otherwise the circle is inside the polygon. It repeats this with each edge of the polygon.

The isCollidingCircleConvex method is called when a circle and a polygon are passed to the is colliding method. This method reverse the order that the objects are passed and redirects these new parameters to isCollidingConvexCircle.

Collisions are resolved by passing the collision info and the bodies to the ResolveSolid method. It starts by checking the collision normal and overlapping vectors are directed correctly if they aren't then their negative replaces them. Next it moves the first body so that the bodies aren't overlapping. Then for each collision point it finds the collision point radii (i.e. the distance between the collision point and each of the centre points of the shapes) and it then finds vectors, which are perpendicular to these radii. The velocity of the two bodies is then got. Then the difference in velocities is then found to get a relative velocity. A collision tangent is calculated to decide the direction in which friction should act. This is found by the following formula:

*Tangent = Velocity – Velocity.Normal * Normal*

The impulse along the collision normal is then calculated using the formula:

*Impulse = (-(1 + Restitution) * Velocity).Normal / Normal.(Normal/Mass)*

Restitution = (Restitution$_1$ + Restitution$_2$) / 2

Next the tangential impulse force is calculated using this formula:

*Impulse = -(1 + Restitution) * Velocity.Tangent / Masses*

The next check is that this impulse is enough to overcome friction. If it is then set the coefficient of friction to dynamic friction coefficient. Next is to check if the impulse is in the right direction and correct it if it isn't. Then finally add the impulses.

Program Code and Explanation

```
using System;

namespace NEA_Physics_Engine
{
	#if WINDOWS
		static class Program
		{
			///Program start
			static void Main(string[] args)
			{
				using (ProgramManager engine =
NEA_Physics_Engine.ProgramManager.Instance)
				{
					engine.Run();
				}
			}
		}
	#endif
}
```

This is the default starting file in c# projects for visual studio. Essentially it begins the program.

RenderManager Code and Explanation

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

using NEA_Physics_Engine.Rendering.Render;

namespace NEA_Physics_Engine.Rendering
{
   //The different layers on the screen (Could potentially be expanded)
        public enum LAYER
        {
                BACK = 0,
                MID,
                FRONT,
                NR_OF_LAYERS
        }

        public sealed class RenderManager
        {
      //Attributes
      //my_SpriteBatch (SpriteBatch):instance of SpriteBatch class used to draw
objects
      //my_Strings (Queue of string, Vector2 pairs): qeueue of strings to be drawn
      //my_RenderObjects (Queue of RenderObject): objects to be drawn
      //my_SpriteFont (SpriteFont): the font for the strings
      //my_PixelWhite (Texture2D): texture for lines
      //my_CircleWhite (Texture2D): texture for circles

      //Methods
      //DrawString: enqueues new string in my_Strings
      //DrawLine: enqueues new line in my_RenderObjects
      //DrawCircle: enqueues new circle in my_RenderObjects
      //Update: run on update and draws the changes that have occured during
the update
      //DrawLayer: draws a layer of the scene

                private SpriteBatch my_SpriteBatch;

      //Set up the queues for rendering objects
                private Queue<KeyValuePair<string, Vector2>> my_Strings;
                private Queue<RenderObject>[] my_RenderObjects;

      //Textures
                private SpriteFont my_SpriteFont;
                private Texture2D my_PixelWhite;
                private Texture2D my_CircleWhite;
```

```
		public void DrawString(string parameter_String, Vector2
parameter_Position = new Vector2())
		{
	//Draw a the given string at the given position
			my_Strings.Enqueue(new KeyValuePair<string,
Vector2>(parameter_String, parameter_Position));
		}

		public void DrawLine(Vector2 parameter_Start, Vector2
parameter_End, int parameter_Width = 1, Color parameter_Colour = new Color(),
LAYER parameter_Layer = LAYER.FRONT)
		{
	//Draw the given line at the given position on the front layer, also in given
colour
			my_RenderObjects[(int)parameter_Layer].Enqueue(new
RenderLine(parameter_Start, parameter_End, parameter_Width,
OBJECT_TYPE.LINE, parameter_Colour));
		}

		public void DrawCircle(Vector2 parameter_Position, float
parameter_Radius, Color parameter_Colour = new Color(), LAYER
parameter_Layer = LAYER.FRONT)
		{
	//Draw a circle of a given radius at a given position on the front layer, also
in given colour
			my_RenderObjects[(int)parameter_Layer].Enqueue(new
RenderCircle(parameter_Radius, parameter_Position, OBJECT_TYPE.CIRCLE,
parameter_Colour));
		}

		public void Update()
		{
			my_SpriteBatch.GraphicsDevice.Clear(Color.Black);
			my_SpriteBatch.Begin();
			{
		//Draw each layer
				for (int i = 0; i < (int)LAYER.NR_OF_LAYERS; i++)
					DrawLayer(my_RenderObjects[i]);

				int j = 0;
		//Draw the strings currently in the string list
				foreach (KeyValuePair<string, Vector2> String in
my_Strings)
				{
					j++;
					my_SpriteBatch.DrawString(my_SpriteFont,
String.Key, (String.Value.X == 0 && String.Value.Y == 0) ? new Vector2(10,
my_SpriteFont.LineSpacing * j) : String.Value, Color.White);
```

```
                        }
                    }
                    my_SpriteBatch.End();

        //Once everything is drawn, clear the rendering queues
                    for (int i = 0; i < (int)LAYER.NR_OF_LAYERS; i++)
                        my_RenderObjects[i].Clear();
                    my_Strings.Clear();
                }

                private void DrawLayer(Queue<RenderObject>
parameter_RenderObjects)
                {
                    foreach (RenderObject renderObject in
parameter_RenderObjects)
                    {
            //Draw each object

                                    if (renderObject.Type == OBJECT_TYPE.LINE)
                                    {
                                        RenderLine line =
(RenderLine)renderObject;
                                        Vector2 lineVector = line.End -
line.Start; //Get the direction vector of the line
                                        int lineLength =
(int)lineVector.Length(); //Get the vector length
                                        lineVector.Normalize();
                                        float perpendicularDotProduct =
lineVector.X * Vector2.UnitX.Y - lineVector.Y * Vector2.UnitX.X;

                //Draw a thin rectangle (i.e. a line)
                                        my_SpriteBatch.Draw(my_PixelWhite,
new Rectangle((int)line.End.X, (int)line.End.Y, lineLength, line.Width), null,
line.Colour, (float)Math.Atan2(perpendicularDotProduct, -
Vector2.Dot(lineVector, Vector2.UnitX)), new Vector2(), SpriteEffects.None,0);
                                    }

                                    if (renderObject.Type ==
OBJECT_TYPE.CIRCLE)
                                    {
                                        RenderCircle circle =
(RenderCircle)renderObject;

                //Draw circle
                                        my_SpriteBatch.Draw(my_CircleWhite,
new Rectangle((int)(circle.Position.X - circle.Radius), (int)(circle.Position.Y -
circle.Radius), (int)(circle.Radius * 2), (int)(circle.Radius * 2)), circle.Colour);
                                    }
```

```
                }
            }

    //Set up the instance
            private static volatile RenderManager my_Instance;
            private static object my_SyncRoot = new Object();
            private RenderManager()
            {
                my_SpriteBatch = new
SpriteBatch(NEA_Physics_Engine.ProgramManager.Instance.GraphicsDevice);
                my_Strings = new Queue<KeyValuePair<string,
Vector2>>();
                my_RenderObjects = new
Queue<RenderObject>[(int)LAYER.NR_OF_LAYERS];
                my_RenderObjects[(int)LAYER.FRONT] = new
Queue<RenderObject>();
                my_RenderObjects[(int)LAYER.MID] = new
Queue<RenderObject>();
                my_RenderObjects[(int)LAYER.BACK] = new
Queue<RenderObject>();

                my_SpriteFont =
ProgramManager.Instance.Content.Load<SpriteFont>("SpriteFont");
                my_PixelWhite =
ProgramManager.Instance.Content.Load<Texture2D>("pixel_white");
                my_CircleWhite =
ProgramManager.Instance.Content.Load<Texture2D>("circle_white");
            }

            public static RenderManager Instance
            {
                get
                {
                    if (my_Instance == null)
                        lock (my_SyncRoot)
                    if (my_Instance == null)
                    {
                        my_Instance = new RenderManager();
                    }

                    return my_Instance;
                }
            }
        }
}
```

The render manager uses a queue to store objects, which need to be rendered. It then uses a library to draw the queued items on each update. It does this by

drawing each layer one after the other. When a new object needs to be rendered it's added to the back of the queue and when an object is rendered it's removed from the front of the queue. Different methods are used for drawing lines, circles and strings.

RenderObject Code and Explanation

```
using System;

using Microsoft.Xna.Framework;

namespace NEA_Physics_Engine.Rendering.Render
{
  //Types of object (could be expanded)
        public enum OBJECT_TYPE
        {
                CIRCLE,
                LINE,
                STRING
        }

        public class RenderObject
        {
     //Methods
     //Type: get/set the type of object
     //Colour: get/set the colour of the object

     //Get object type and colour
                public OBJECT_TYPE Type {get; protected set;}
                public Color Colour {get; protected set;}

                protected RenderObject(OBJECT_TYPE parameter_Type, Color
parameter_Colour)
                {
                        Type = parameter_Type;
                        if (parameter_Colour.A != 0)
                                Colour = parameter_Colour;
                        else
                                Colour = Color.White; //If no colour is given, draw in
white
                }
        }
}
```

The RenderObject is the superclass of the RenderLine and RenderCircle classes. It contains an enumerator for the 3 types of render object. There are also some getters and setters for the object type and the colour. The type is set by the parameter passed and if the colour isn't zero then the colour is the parameter passed otherwise the colour is set to white.

RenderLine Code and Explanation

```
using Microsoft.Xna.Framework;

namespace NEA_Physics_Engine.Rendering.Render
{
        public class RenderLine : RenderObject
        {
    //Methods
    //Start: get/set the start point of the line
    //End: get/set the end point of the line
    //Width: get/set the width of the line

    //Get details for line
                public RenderLine(Vector2 parameter_Start, Vector2
parameter_End, int parameter_Width, OBJECT_TYPE parameter_Type, Color
parameter_Color) : base(parameter_Type, parameter_Color)
                {
                        Start = parameter_Start;
                        End = parameter_End;
                        Width = parameter_Width;
                }

                public Vector2 Start {get; private set;}
                public Vector2 End {get; private set;}
                public int Width {get; private set;}
        }
}
```

A small class used for setting up lines that are to be rendered. It has a start point, end point and a width.

RenderCircle Code and Explanation

```
using Microsoft.Xna.Framework;

namespace NEA_Physics_Engine.Rendering.Render
{
        public class RenderCircle : RenderObject
        {
    //Methods
    //Radius: get/set the radius of the circle
    //Position: get/set the position of the circle

    //Get details fore the circle
                public RenderCircle(float parameter_Radius, Vector2
parameter_Position, OBJECT_TYPE parameter_Type, Color parameter_Color) :
base(parameter_Type, parameter_Color)
                {
```

```
                Radius = parameter_Radius;
                Position = parameter_Position;
            }
            public float Radius {get; private set;}
            public Vector2 Position {get; private set;}
        }
}
```

A small class used for setting up circles that are to be rendered. Has a radius and a central position.

mathsUtility Code and Explanation

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;

namespace NEA_Physics_Engine.Utilities
{
   public class mathsUtility
   {
      //Methods
      //Angle: find the angle between 2 vectors
      //leftPerpendicular: find the perpendicular vector 90 degrees to left of given
vector
      //rightPerpendicular: find the perpendicular 90 degrees to the right of given
vector
      //LineLineIntersection: find whether two vectors intersect
      //CrossProduct: calculate the vector cross product of two vectors
      //PolygonCenter: calculate the central point of a polygon
      //PolygonInertia: calculate the inertia of a polygon
      //PolygonVolume: approximate the volume of a polygon
      //CircleVolume: calculate the volume of a circle

      //Find angle between vectors
      public static double Angle(Vector2 matrix1, Vector2 matrix2)
      {
         matrix1.Normalize();
         matrix2.Normalize();
         double angle = System.Math.Atan2(matrix2.Y - matrix1.Y, matrix2.X -
matrix1.X);
         /// if angle is tiny then don't calculate
         if (angle < 0.0001)
            return 0;
         else
            return angle;
      }

      //Find the perpendicular vector
```

```
    public static Vector2 leftPerpendicular(Vector2 matrix)
    {
        return new Vector2(-matrix.Y, matrix.X);
    }

    public static Vector2 rightPerpendicular(Vector2 matrix)
    {
        return new Vector2(matrix.Y, -matrix.X);
    }

    //Find whether vectors intersect
    public static bool LineLineIntersection(Vector2 start1, Vector2 end1,
Vector2 start2, Vector2 end2, out Vector2 intersectionPoint)
    {
        //Set intersection point to none
        intersectionPoint = Vector2.Zero;
        float denominator = ((end1.X - start1.X) * (end2.Y - start2.Y)) - ((end1.Y -
start1.Y) * (end2.X - start2.X));

        //  AB & CD are parallel
        if (denominator == 0)
            return false;


        float numerator = ((start1.Y - start2.Y) * (end2.X - start2.X)) - ((start1.X -
start2.X) * (end2.Y - start2.Y));

        float r = numerator / denominator;

        float numerator2 = ((start1.Y - start2.Y) * (end1.X - start1.X)) - ((start1.X -
start2.X) * (end1.Y - start1.Y));

        float s = numerator2 / denominator;

        if ((r < 0 || r > 1) || (s < 0 || s > 1))
            return false;

        // Find intersection point
        intersectionPoint = new Vector2();
        intersectionPoint.X = start1.X + (r * (end1.X - start1.X));
        intersectionPoint.Y = start1.Y + (r * (end1.Y - start1.Y));

        return true;
    }

    //Calculate vector cross product of two vectors
    public static float CrossProduct(Vector2 matrix1, Vector2 matrix2)
    {
        return (matrix1.X * matrix2.Y) - (matrix1.Y * matrix2.X);
```

```
    }

    //Find the center of the polygon
    public static Vector2 PolygonCenter(Vector2[] parameter_Vertices, float
parameter_Volume)
    {
        Vector2 center = new Vector2(); //Center vector

        //Find center using the centroid formula
        for (int i = 0; i < parameter_Vertices.Length; i++)
        {
            Vector2 vertex1 = parameter_Vertices[i];
            Vector2 vertex2 = parameter_Vertices[i + 1 < parameter_Vertices.Length
? i + 1 : 0];
            center.X += (vertex1.X + vertex2.X) * (vertex1.X * vertex2.Y - vertex2.X *
vertex1.Y);
            center.Y += (vertex1.Y + vertex2.Y) * (vertex1.X * vertex2.Y - vertex2.X *
vertex1.Y);
        }

        center.X *= 1 / (6 * parameter_Volume);
        center.Y *= 1 / (6 * parameter_Volume);
        return center;
    }

    public static float PolygonInertia(Vector2[] parameter_Vertices, float
parameter_Mass)
    {
        //Polygon inertia, found by taking moments around each vertex
        float sum1 = 0.0f;
        float sum2 = 0.0f;
        for (int i = 0; i < parameter_Vertices.Length; i++)
        {
            Vector2 v1 = parameter_Vertices[i];
            Vector2 v2 = parameter_Vertices[(i + 1) % parameter_Vertices.Length];
            v2.Normalize();

            float a = Vector2.Dot(v2, v1);
            float b = Vector2.Dot(v1, v1) + Vector2.Dot(v1, v2) + Vector2.Dot(v2,
v2);

            sum1 += a * b;
            sum2 += a;
        }
        return Math.Abs((parameter_Mass * sum1) / (6.0f * sum2));
    }

    public static float PolygonVolume(Vector2[] parameter_Vertices)
    {
```

```
        float area = 0;
        int j = parameter_Vertices.Length - 1;

        for (int i = 0; i < parameter_Vertices.Length; i++)
        {
          //Approximatee area by finding the area created between 2 points and
adding the area (+ive * -ive gives a -ive area)
          area += (parameter_Vertices[j].X + parameter_Vertices[i].X) *
(parameter_Vertices[j].Y - parameter_Vertices[i].Y);
          j = i;
        }

        //Return the modulus of the area of the area
        return Math.Abs(area * 0.5f);
      }

      public static float CircleVolume(float parameter_Radius)
      {
        float area;

        //3.142*Radius^2
        area = (float)Math.PI * (parameter_Radius * parameter_Radius);

        //Return the modulus of the area
        return Math.Abs(area * 0.5f);
      }
    }
}
```

The mathsUtility class contains lots of mathematical formulae that are used throughout the program. The Angle method calculates the angle between two vectors that are passed as parameters, an alternate method was used as inverse cosine has multiple solutions. The leftPerpendicular and rightPerpendicular methods return perpendicular vectors to the ones passed in, it does that by dividing one of the two dimensions by -1.

The LineLineIntersection method returns whether two vectors intersect. It calculates a denominator based on the vector dimensions given. If the denominator is 0 then the vectors are parallel and don't intersect. Otherwise 2 numerators are calculated based on the vector dimensions. It then finds two fractions based on dividing the numerators by the denominator. If either of the fractions are negative or top heavy (i.e. greater than 1) then the vectors don't intersect. Otherwise it calculates the intersection point.

The CrossProduct method calculates the vector cross product between two vectors and returns the result. The PolygonCenter method finds the centre of a polygon given. It works on a basis of geometric composition which is essentially splitting the complex polygon into simpler polygons finding their centres and then finding the centre of those.

The PolygonInertia method finds the inertia of a polygon passed. The polygon inertia is calculated using the following formula:

$Inertia = (Mass / 6) * ((\Sigma(x_i^2 + y_i^2 + x_i x_{i+1} + y_i y_{i+1} + x_{i+1}^2 + y_{i+1}^2)(x_i y_{i+1} - x_{i+1} y_i))/(\Sigma(x_i y_{i+1} - x_{i+1} y_i)))$

The sums are computed iteratively.

The PolygonVolume method is used to calculate the volume of the polygon passed as a parameter. There is no exact formula for the volume of any polygon so a heuristic approach was used. The method approximates the area by draw squares between adjacent vertices and finding their area. By adding up these areas, an estimate is found for the polygon's area. The CircleVolume method is used to calculate the volume of the circle passed as a parameter. This makes use of the formula:

$Area = \pi * Radius^2$

Controls

Sandbox Mode

Input	Operation
P	Toggle polygon drawing mode
O	Toggle circle drawing mode
C	Clear all objects
X	Toggle controls panel
M	Toggle object debug
Q	Toggle whether drawn objects will be static
G	Toggle whether drawn objects will be affected by gravity
NUMPAD 1	Change material to solid
NUMPAD 2	Change material to ice
NUMPAD 3	Change material to wood
NUMPAD 4	Change material to metal
NUMPAD 5	Change material to rubber
1	Show scenario 1
2	Show scenario 2
3	Show scenario 3
4	Show scenario 4
5	Show scenario 5
6	Show scenario 6
7	Show scenario 7
8	Show scenario 8
9	Show scenario 9
F1	Show scenario 10
W	Make gravity vector more upwards
A	Make gravity vector more to the left

S	Make gravity vector more downwards
D	Make gravity vector more to the right
Enter	Reset gravity vector

This mode is for watching how the engine functions with collisions.

Polygon Drawing Mode

Input	**Operation**
Left Mouse Button	Place point (When valid)
P	Toggle polygon drawing mode (Creates current polygon if 3 or more points placed)

Use this mode to draw custom convex polygons. The cursor will be green when you can place a point and red when you can't.

Circle Drawing Mode

Input	**Operation**
Left Mouse Click and Drag	Set circle radius
O	Toggle circle drawing mode (Creates current circle if radius set)

Use this mode to draw circles.

Testing

Test No.	Test	Purpose	Test Type	Expected Result	Pass/Fail
1	Starting the program.	Check program runs and loads up the correct content.	Typical	Sandbox mode scene should be loaded with control panel open.	Pass
2	Press P when polygon drawing mode isn't engaged.	Check that the polygon drawing mode can be toggled.	Typical	String indicating that polygon drawing mode should appear as well as a cursor change.	Pass
3	Press P when polygon drawing mode is engaged when no points placed.	Check that the polygon drawing mode can be toggled.	Erroneous	Polygon drawing mode should disengage. Indicated by string vanishing.	Pass
4	In polygon drawing mode, move cursor into a valid position.	Check that the system recognises valid places for a vertex to be placed.	Typical	Cursor should be green.	Pass
5	In polygon drawing mode, move cursor into an invalid position.	Check that the system recognises invalid places for a vertex to be placed.	Erroneous	Cursor should be red.	Pass
6	In polygon drawing mode, click when cursor is green.	Check that vertices can be placed in valid locations.	Typical	Point should be placed, indicated by a circle.	Pass
7	In polygon drawing mode, click when cursor is red.	Check that vertices can't be placed in invalid locations.	Erroneous	Point shouldn't be placed.	Pass

8	In polygon drawing mode, press P when 2 or less points have been placed.	Check that 1-sided or 1-dimensional shapes can't be drawn.	Erroneus	Polygon mode should disengage and no polygon should be drawn.	Pass
9	In polygon drawing mode, press P when 3 points have been placed.	Check that valid polygons can be drawn.	Boundary	Polygon mode should disengage and a polygon should be created with straight edges connecting adjacent vertices.	Pass (see figure 1)
10	In polygon drawing mode, press P when more than 3 points have been placed.	Check that valid polygons can be drawn.	Typical	Polygon mode should disengage and a polygon should be created with straight edges connecting adjacent vertices.	Pass (see figure 1)
11	Press O when circle drawing mode isn't engaged.	Check that the circle drawing mode can be toggled.	Typical	String indicating that circle drawing mode should appear.	Pass
12	Press O when circle drawing mode is engaged and no circle has been drawn.	Check that the circle drawing mode can be toggled.	Erroneous	Circle drawing mode should disengage and no circle drawn.	Pass
13	Click and drag when in circle drawing mode.	Check that the radius of a circle can be set based on user input.	Typical	Should set radius of circle with centre at the clicked point, indicated by a green circle.	Pass
14	Press O when circle	Check that circles can	Typical	Circle drawing	Pass (see figure 2)

	drawing mode is engaged and a circle has been drawn.	be drawn.		mode should disengage and a circle should be created.	
15	Press ESC.	Check program can be exited.	Typical	Program should close.	Pass
16	Press X when control help is on.	Check that control help can be toggled.	Typical	Control help should turn off (controls shouldn't be displayed)	Pass
17	Press X when control help is off.	Check that control help can be toggled and that controls are displayed correctly.	Typical	Control help should turn on (controls should be displayed)	Pass
18	Press Q when isStatic is false.	Check whether the user can toggle whether the next object will be static or not.	Typical	isStatic should be true and gravityAffects should be false indicated by strings in the drawing modes.	Pass
19	Press Q when isStatic is true.	Check whether the user can toggle whether the next object will be static or not.	Typical	isStatic should be false indicated by a string in the drawing modes.	Pass
20	Press G when gravityAffects is false.	Check whether the user can toggle whether the next object will be affected by gravity or not.	Typical	gravityAffects should be true indicated by a string in the drawing modes.	Pass

21	Press G when gravityAffects is true.	Check whether the user can toggle whether the next object will be affected by gravity or not.	Typical	gravityAffects should be false indicated by a string in the drawing modes.	Pass
22	Draw a polygon when isStatic is true and gravityAffects is false.	Check whether static polygons can be created.	Typical	Polygon should stay where it is and be unaffected by gravity or impulses from collision.	Pass (see figures 3 and 4)
23	Draw a polygon when isStatic is false and gravityAffects is true.	Check whether polygons which are affected by all physics can be created.	Typical	Polygon should be affected by gravity and impulses from collision.	Pass (see figure 3)
24	Draw a polygon when isStatic is false and gravityAffects is false.	Check whether polygons which are affected by collisions but not gravity can be created.	Typical	Polygon shouldn't be affected by gravity but should be affected by impulses from collision.	Pass
25	Draw a circle when isStatic is true and gravityAffects is false.	Check whether static circles can be created.	Typical	Circle should stay where it is and be unaffected by gravity or impulses from collision.	Pass (see figures 3 and 4)
26	Draw a circle when isStatic is false and gravityAffects is true.	Check whether circles which are affected by all	Typical	Circle should be affected by gravity and impulses from	Pass (see figure 4)

		physics can be created.		collision.	
27	Draw a circle when isStatic is false and gravityAffects is false.	Check whether circles which are affected by collisions but not gravity can be created.	Typical	Circle shouldn't be affected by gravity but should be affected by impulses from collision.	Pass
28	When debug is off, press M and click on the centre circle of an object.	Check that debug can be toggled.	Typical	Details about the object (debug) should be displayed.	Pass
29	Press M when debug is on.	Check that debug can be toggled.	Typical	Debug should turn off.	Pass
30	Let a non-static, gravity-affected polygon collide with the floor (or another static object).	Check that collisions between non-static, gravity-affected polygons and static objects are detected and resolved as intended.	Typical	The polygon should be stopped, depending on the height dropped from it should bounce. When the polygon stops bouncing it should oscillate slightly back and forth due to SHM. No effect should be seen on the static object.	Pass (see figure 3)
31	Let non-static, gravity-affected circle collide with the floor (or another static object).	Check that collisions between non-static, gravity-affected circles and static objects are detected	Typical	The circle should be stopped, depending on the height dropped from it should bounce. No affect should	Pass (see figure 4)

		and resolved as intended.		be seen on the static object.	
32	Let two non-static, gravity-affected polygons collide.	Check that collisions between non-static, gravity-affected polygons are detected and resolved as intended.	Typical	The two polygons should bounce off one another and some rotation should occur. When the polygons stop bouncing they should oscillate slightly due to SHM.	Pass (see figure 5)
33	Let two non-static, gravity-affected circles collide.	Check that collisions between non-static, gravity-affected circles are detected and resolved as intended.	Typical	The two circles should bounce off one another.	Pass (see figure 6)
34	Let a non-static, gravity-affected circle collide with a non-static, gravity-affected polygon.	Check that collisions between non-static, gravity-affected polygons and non-static, gravity-affected circles are detected and resolved as intended.	Typical	The two objects should bounce off one another and some rotation should occur. When the polygon stops bouncing they should oscillate slightly due to SHM.	Pass (see figure 7)
35	Let a non-static, gravity-affected circle collide with a non-static, not gravity-	Check that collisions between, non-static, gravity-affected	Typical	The two objects should bounce off one another. The gravity-	Pass (see figure 8)

	affected circle.	circles and non-static, not gravity-affected circles are detected and resolved as intended.		affected circle should end up at rest on the floor and the not gravity-affected circle will likely end up floating somewhere in the air.	
36	Let a non-static, gravity-affected circle collide with a non-static, not gravity-affected polygon.	Check that collisions between, non-static, gravity-affected circles and non-static, not gravity-affected polygons are detected and resolved as intended.	Typical	The two objects should bounce off one another. The gravity-affected circle should end up at rest on the floor and the not gravity-affected polygon will rotate, and eventually will come to rest - likely floating somewhere in the air.	Pass (see figure 9)
37	Let a non-static, gravity-affected polygon collide with a non-static, not gravity-affected circle.	Check that collisions between, non-static, gravity-affected polygons and non-static, not gravity-affected circles are detected and resolved as intended.	Typical	The two objects should bounce off one another. The gravity-affected polygon should end up oscillating on the floor (due to SHM) and the not gravity-affected circle will likely end up floating somewhere in the air.	Pass (see figure 10)

38	Let a non-static, gravity-affected polygon collide with a non-static, not gravity-affected polygon.	Check that collisions between, non-static, gravity-affected polygons and non-static, not gravity-affected polygons are detected and resolved as intended.	Typical	The two objects should bounce off one another. The gravity-affected polygon should end up oscillating on the floor (due to SHM) and the not gravity-affected polygon will rotate, and eventually will come to rest - likely floating somewhere in the air.	Pass (see figure 11)
39	Press NUMPAD1 and draw any object.	To check that the material of objects can be changed.	Typical	The material should be set to 1 (SOLID), this should be indicated by a string in the drawing modes and the object should have a white outline.	Pass
40	Press NUMPAD2 and draw any object.	To check that the material of objects can be changed.	Typical	The material should be set to 2 (ICE), this should be indicated by a string in the drawing modes and the object should have a light blue outline.	Pass
41	Press NUMPAD3 and draw any	To check that the material of	Typical	The material should be set to 3 (WOOD),	Pass

	object.	objects can be changed.		this should be indicated by a string in the drawing modes and the object should have a brown outline.	
42	Press NUMPAD4 and draw any object.	To check that the material of objects can be changed.	Typical	The material should be set to 4 (METAL), this should be indicated by a string in the drawing modes and the object should have a grey outline.	Pass
43	Press NUMPAD5 and draw any object.	To check that the material of objects can be changed.	Typical	The material should be set to 5 (RUBBER), this should be indicated by a string in the drawing modes and the object should have a orange outline.	Pass
44	Press 1	To check that pre-sets can be loaded and test that friction and damping work as expected.	Typical	The 1st scenario should be loaded. The control panel should turn off. The ball should roll down the ramp and fall well short of the wall. An explanation should be displayed.	Pass (see figure 12)
45	Press 2	To check	Typical	The 2nd	Pass (see

		that pre-sets can be loaded and to check that damping and conservation of energy work as intended.		scenario should be loaded. The control panel should turn off. The ball should roll down one side and begin rolling up the other, it should then roll back down from a lower point. This should repeat until the ball comes to rest in the middle. An explanation should be displayed.	figure 13)
46	Press 3	To check that pre-sets can be loaded.	Typical	The 3rd scenario should be loaded. The control panel should turn off. A set of yellow points should be displayed. An explanation should be displayed.	Pass (see figure 14)
47	Press 4	To check that pre-sets can be loaded.	Typical	The 4th scenario should be loaded. The control panel should turn off. Two concentric yellow circles should be displayed. An explanation	Pass (see figure 15)

				should be displayed.	
48	Press 5	To check that pre-sets can be loaded and a secondary test of collision.	Typical	The 5th scenario should be loaded. The control panel should turn off. The ball should roll down the ramp into the other ball causing the second ball to roll. An explanation should be displayed.	Pass (see figure 16)
49	Press 6	To check that pre-sets can be loaded and to check that a larger mass requires more of an impulse to move an equal distance (i.e. conservation of momentum and friction).	Typical	The 6th scenario should be loaded. The control panel should turn off. Two collisions should occur. The larger ball should stop closer to its start position than the smaller ball. An explanation should be displayed.	Pass (see figure 17)
50	Press 7	To check that pre-sets can be loaded and to check that a larger velocity results in a larger impulse (i.e. conservation	Typical	The 7th scenario should be loaded. The control panel should turn off. Two collisions should occur. The ball on the top layer	Pass (see figure 18)

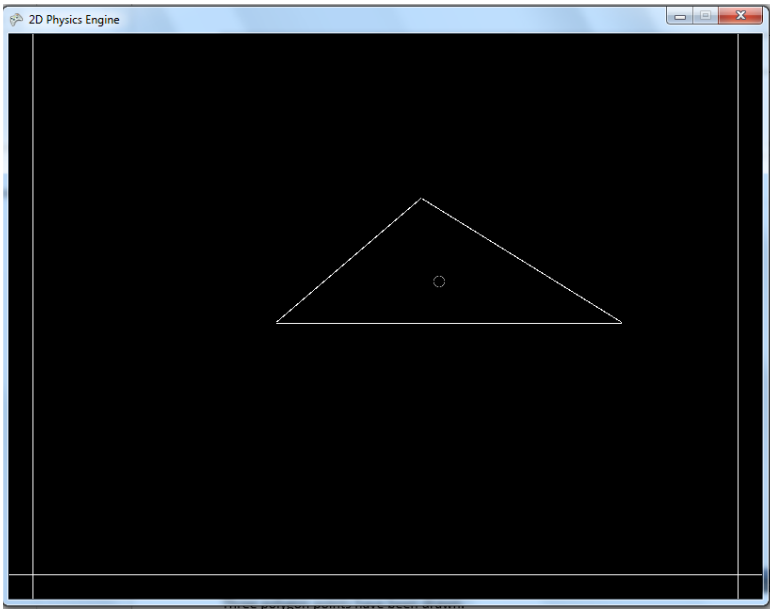
		of momentum).		should roll further than the ball on the bottom layer. An explanation should be displayed.	
51	Press 8	To check that pre-sets can be loaded and test that the system can deal with a large number of collisions at once.	Typical	The 8th scenario should be loaded. The control panel should turn off. A large ball should collide with many smaller circles which are unaffected by gravity. An explanation should be displayed.	Pass (see figure 19)
52	Press 9	To check that pre-sets can be loaded and test whether the material of an object has any affect on its frictional coefficients and density.	Typical	The 9th scenario should be loaded. The control panel should turn off. There should be 2 balls, one should be the basic solid material and the other one should be the currently selected material. Presuming the second ball is not also the basic solid material, it should roll a different	Pass (see figure 20)

				distance. An explanation should be displayed.	
53	Press F1	To check that pre-sets can be loaded and test whether the material of an object has any affect on its restitution coefficients and density.	Typical	The 10th scenario should be loaded. The control panel should turn off. There should be 5 balls, one of each material. Each one should bounce to different heights and therefore bounce for a different amount of time. An explanation should be displayed.	Pass (see figure 21)
54	Press 0	To check the sandbox mode can be loaded.	Typical	The sandbox mode should be loaded.	Pass
55	Press P during a pre-set other than pre-set 3.	To check that the user can't add any objects to pre-sets other than where it's required.	Erroneous	Polygon drawing mode shouldn't engage, to prevent tampering with the scenarios.	Pass
56	Press O during a pre-set other than pre-set 4.	To check that the user can't add any objects to pre-sets other than where it's required.	Erroneous	Circle drawing mode shouldn't engage, to prevent tampering with the scenarios.	Pass

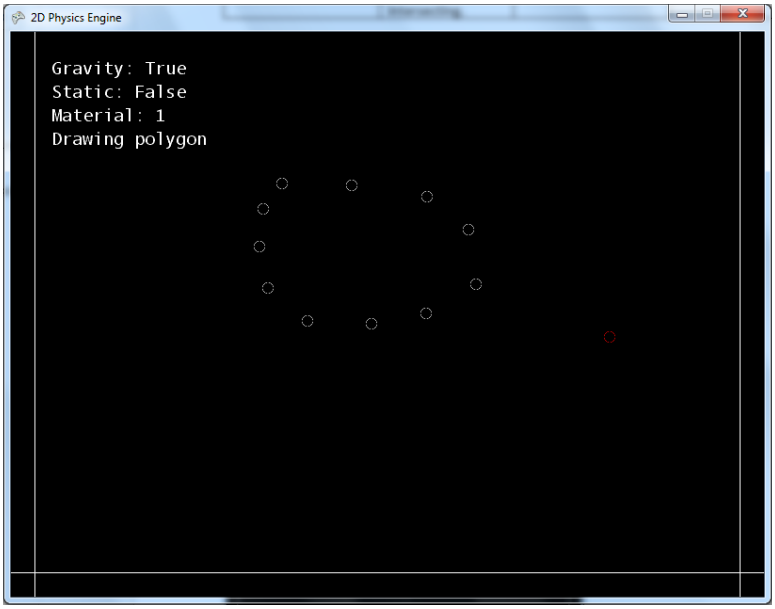
57	Draw two objects which encroach on one another's space.	To check whether the system can resolve two objects which are intersecting.	Erroneous	The two objects should be separated (i.e. resolved).	Pass (see figure 22)
58	Press W	To test that the gravity vector can be manipulated.	Typical	The gravity vector should decrease in Y value (gravity acts more upwards).	Pass
59	Press S	To test that the gravity vector can be manipulated.	Typical	The gravity vector should increase in Y value (gravity acts more downwards).	Pass
60	Press A	To test that the gravity vector can be manipulated.	Typical	The gravity vector should decrease in X value (gravity acts more to the left).	Pass
61	Press D	To test that the gravity vector can be manipulated.	Typical	The gravity vector should increase in X value (gravity acts more to the right).	Pass
62	Press Enter	To test that the gravity vector can be manipulated.	Typical	The gravity vector should reset to (0, 20)	Pass

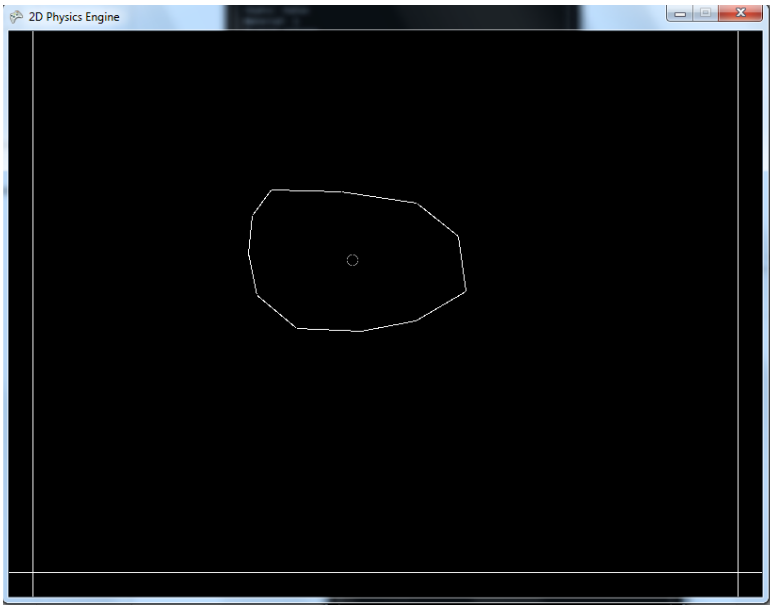
Figure 1

Three polygon points were placed.



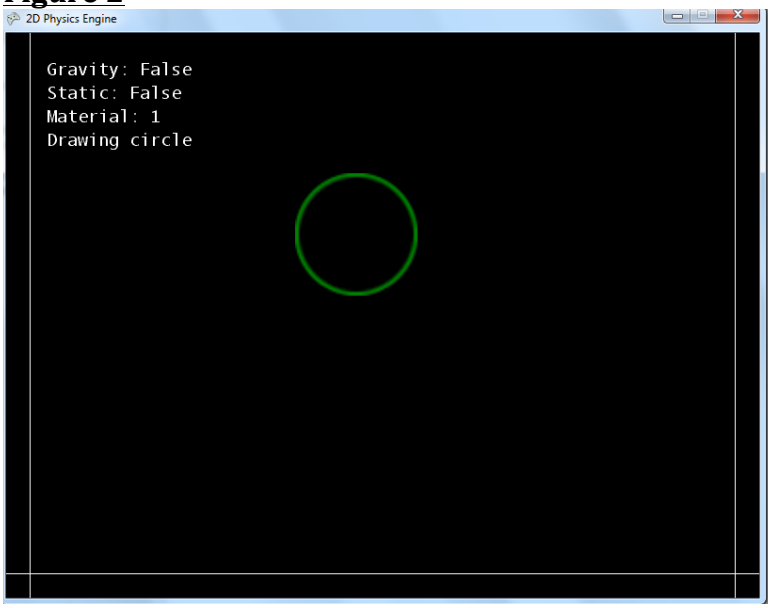
And, when the polygon drawing mode was turned off, a polygon was created by drawing straight edges between the adjacent points.



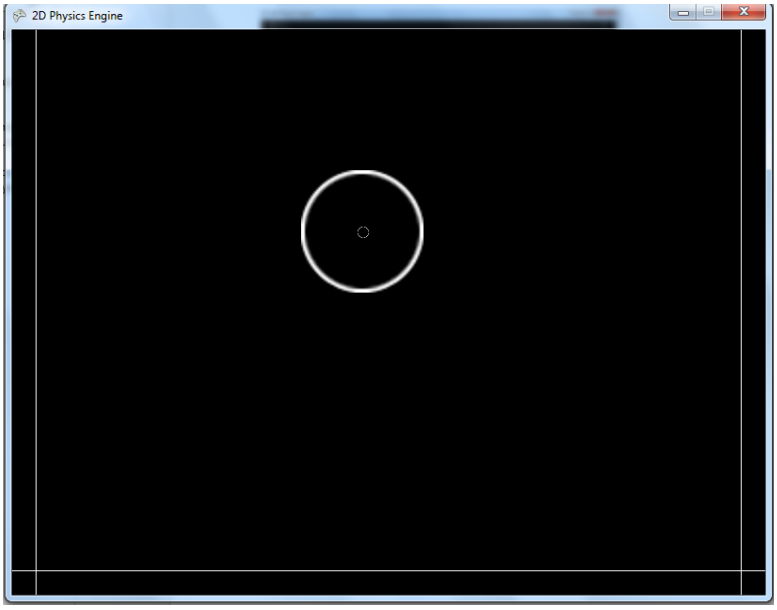


A repeated test with more points.

Figure 2

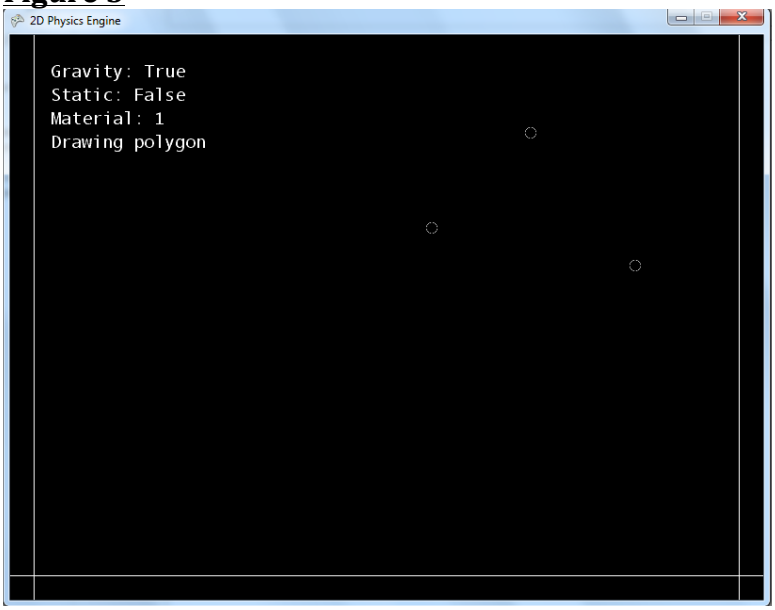


The circle radius was set.

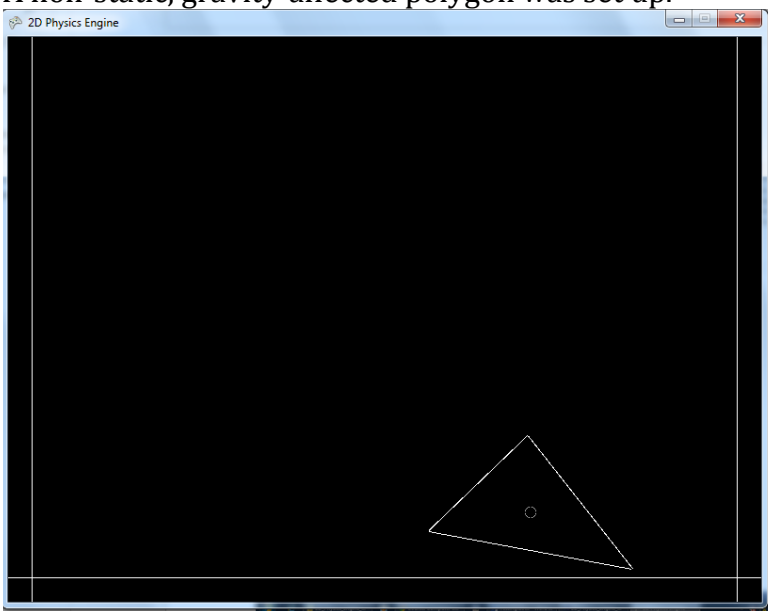


And, when the circle drawing mode was turned off, a circle was drawn.

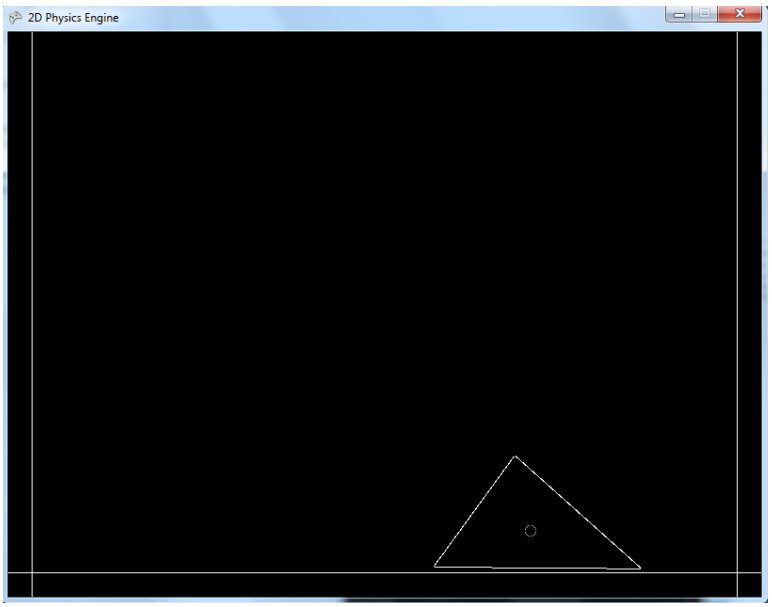
Figure 3



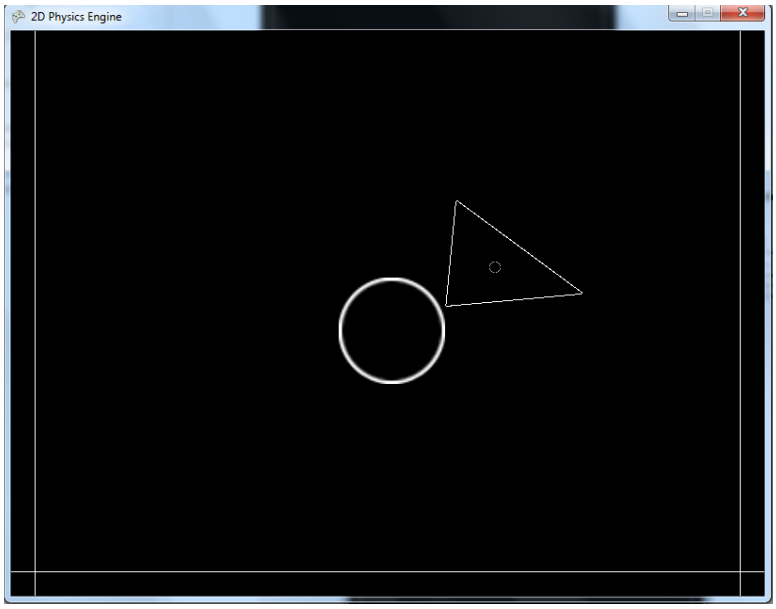
A non-static, gravity-affected polygon was set up.



Gravity pulled it down and it collided with the floor (a static polygon).

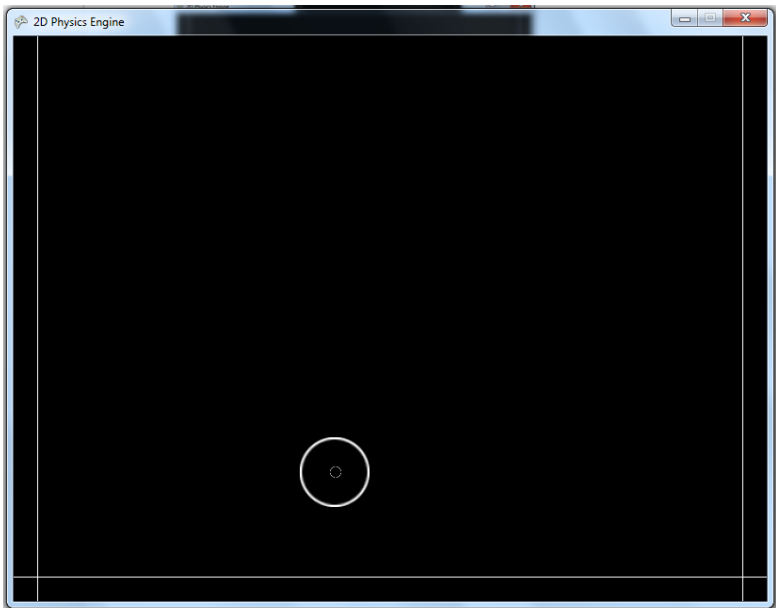


And eventually came to a point where it oscillated a small amount. The static polygon (the floor) was unaffected.

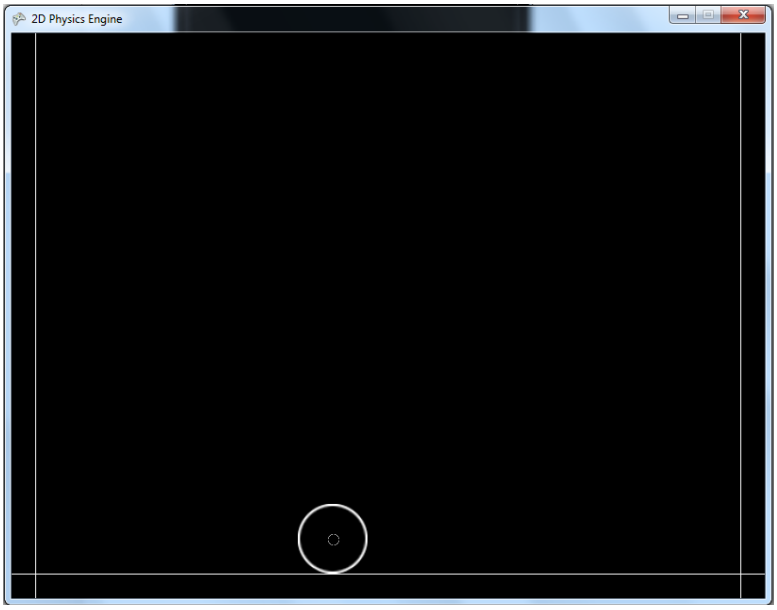


The result when colliding with a static circle, again the polygon bounces off and the static circle is unaffected.

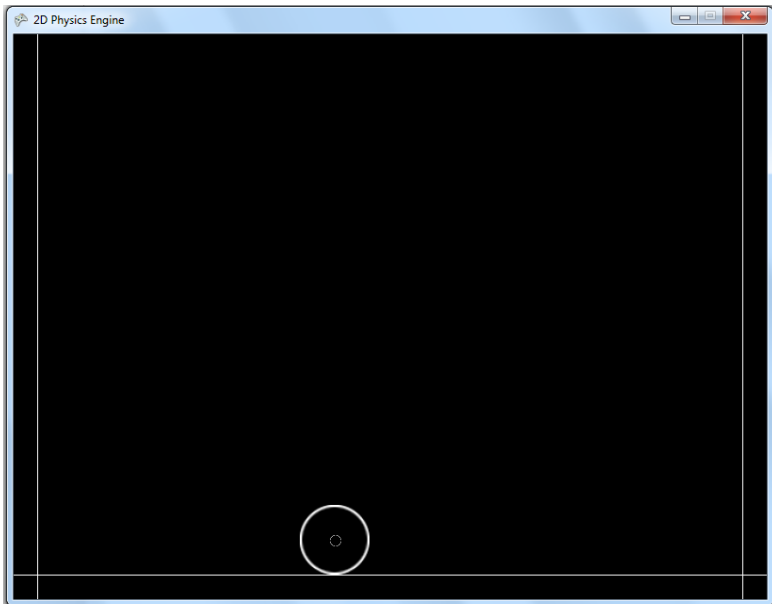
Figure 4



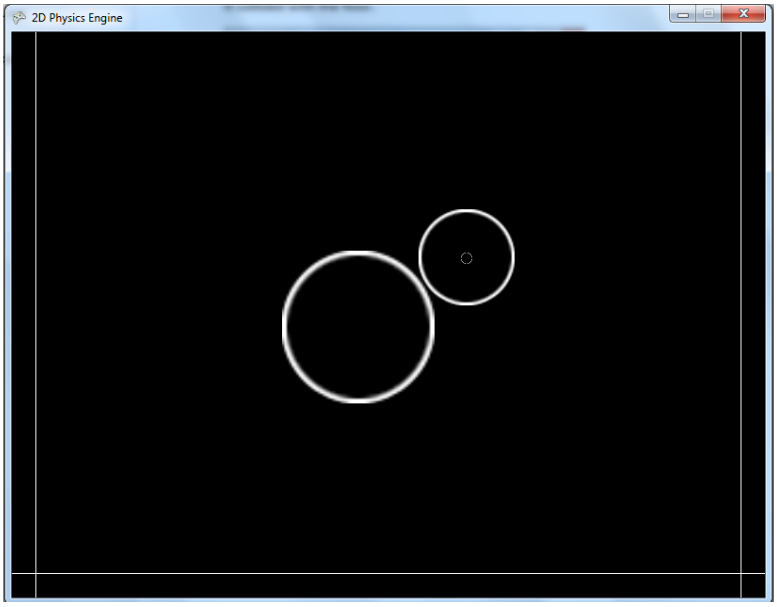
A circle was drawn.



Gravity pulled it down and it collided with the floor (a static polygon).

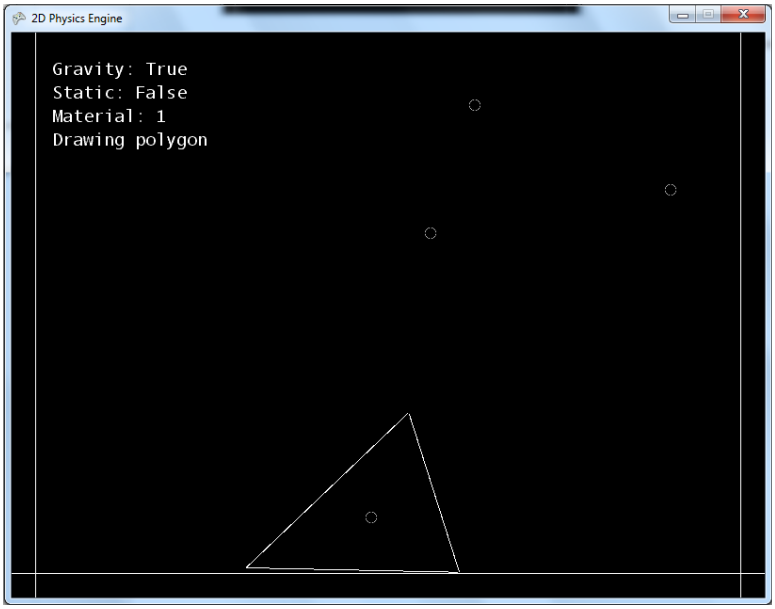


After bouncing a few times, it came to rest. The floor was unaffected.

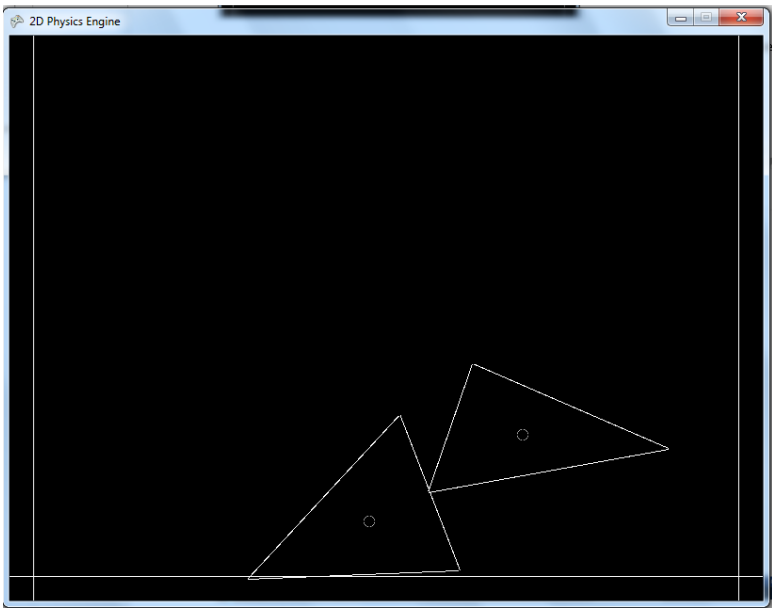


A similar result was found with a static circle. The non-static, gravity-affected circle bounced off. The static circle was unaffected.

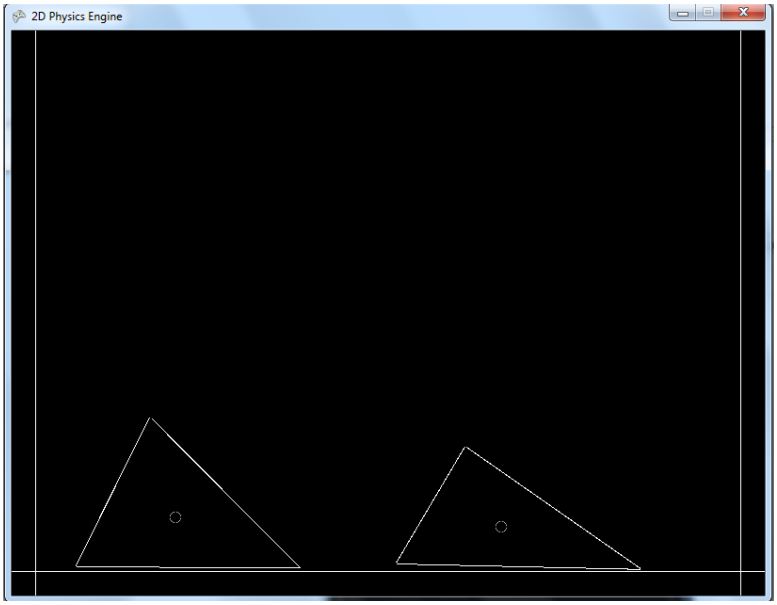
Figure 5



A non-static, gravity-affected polygon was drawn and allowed to come to its oscillating state on the floor. A second non-static, gravity-affected polygon was then drawn.

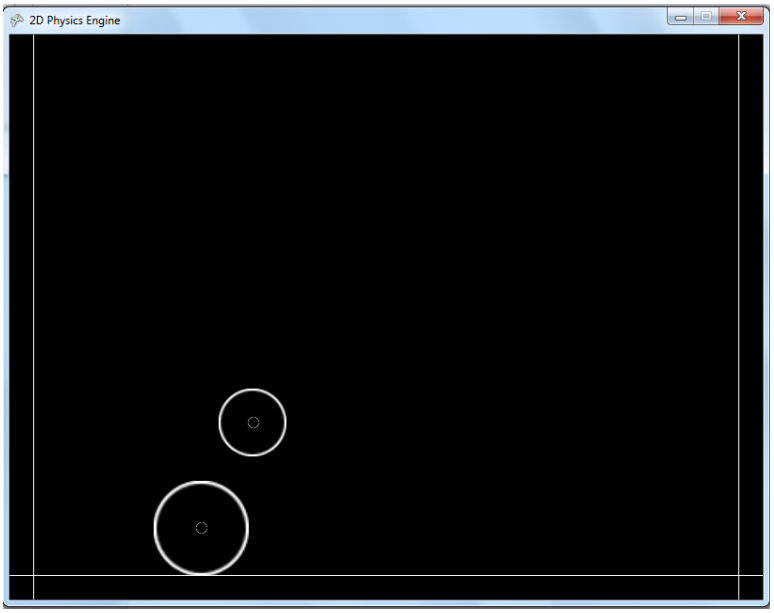


The two polygons collided and bounced off each other.

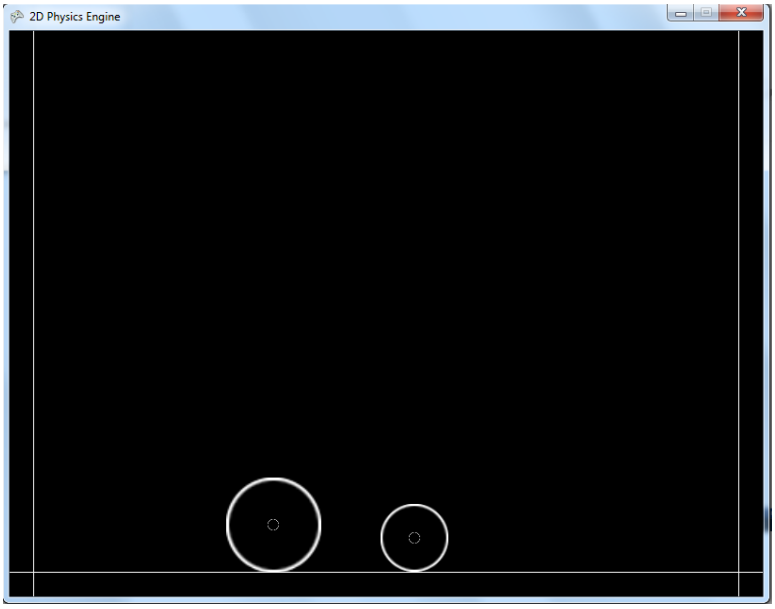


Eventually the two polygons came to their oscillatory states. As you can see the polygons rotated due to the force applied, this is due to the torque caused.

Figure 6

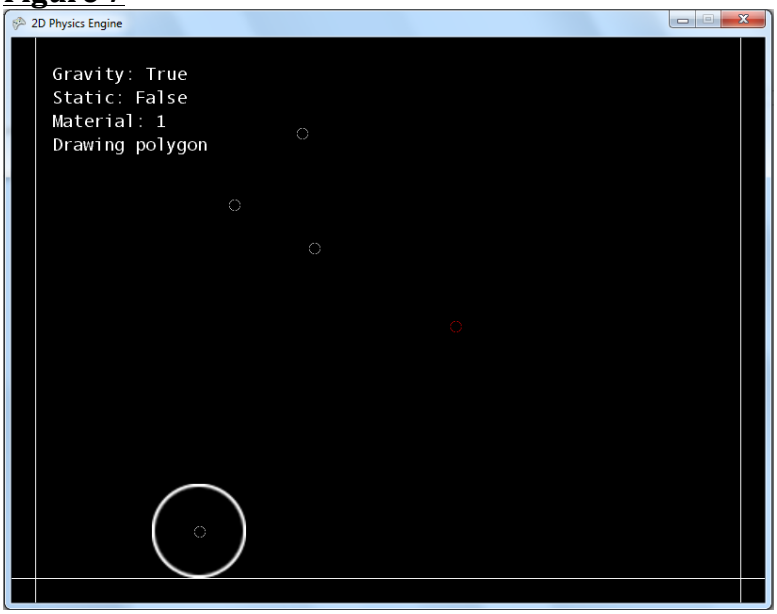


Two non-static, gravity-affected circles were drawn and allowed to collide.

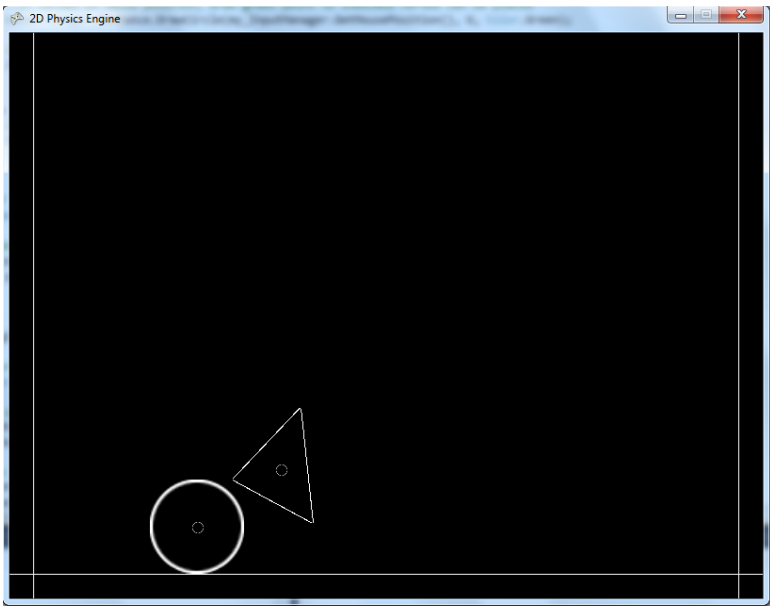


They bounced off one another and eventually came to rest.

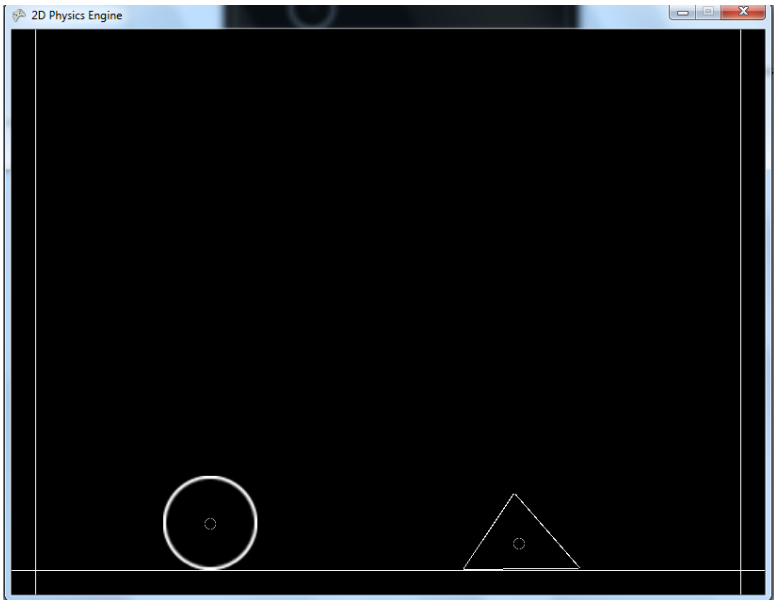
Figure 7



A non-static, gravity-affected circle was drawn and allowed to come to rest. A polygon was then drawn.

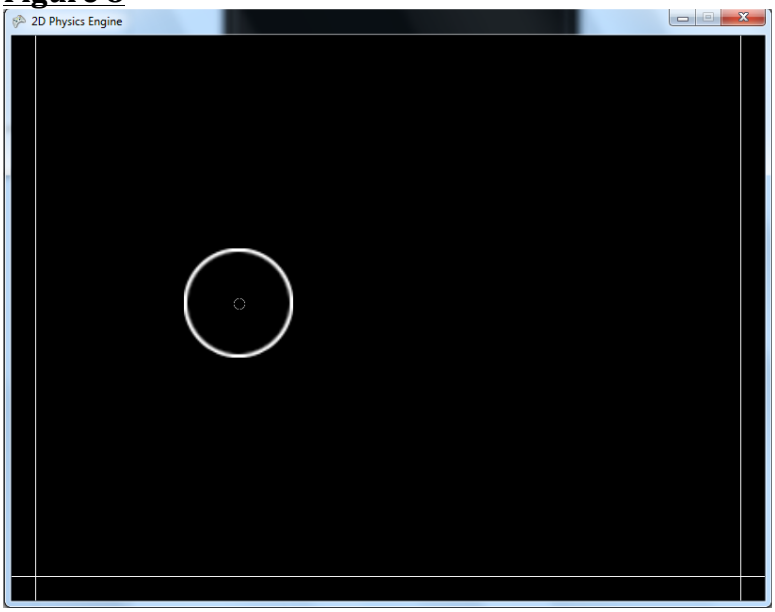


They collided and bounced off each other.

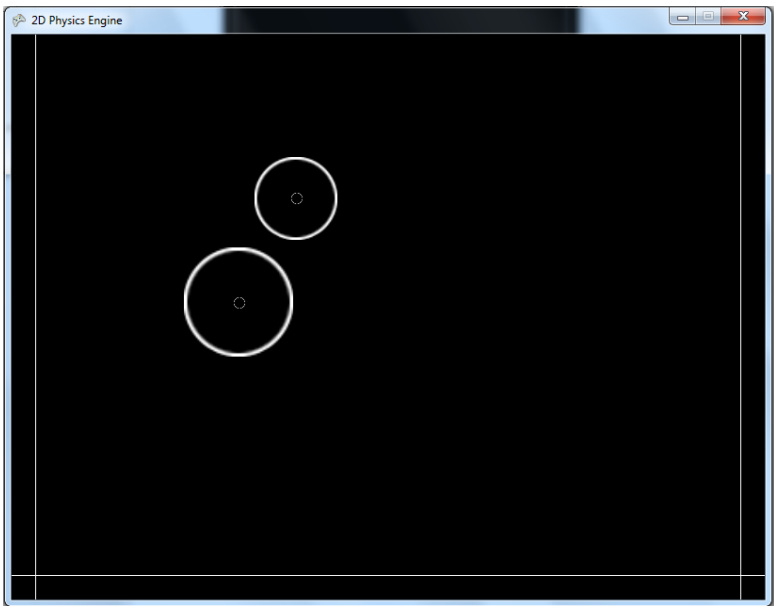


Eventually, the circle came to rest and the polygon reached its oscillatory state.

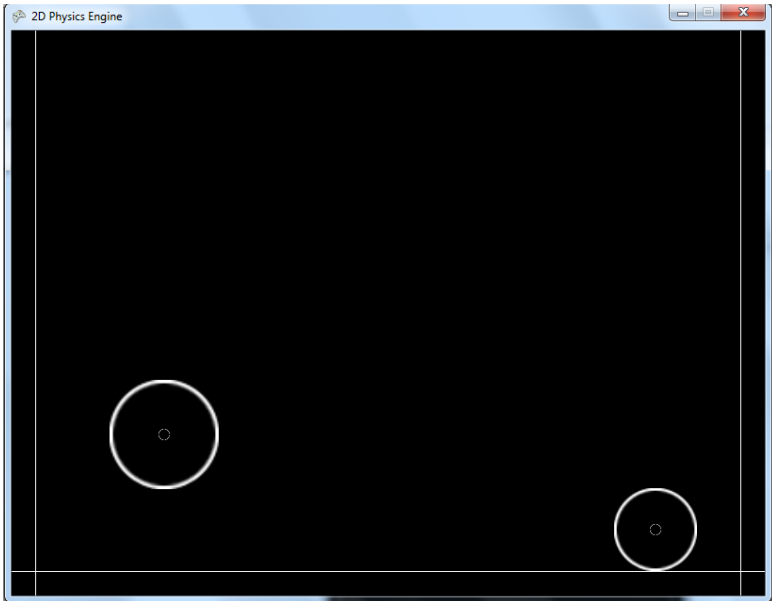
Figure 8



A non-static, not gravity-affected circle was drawn.

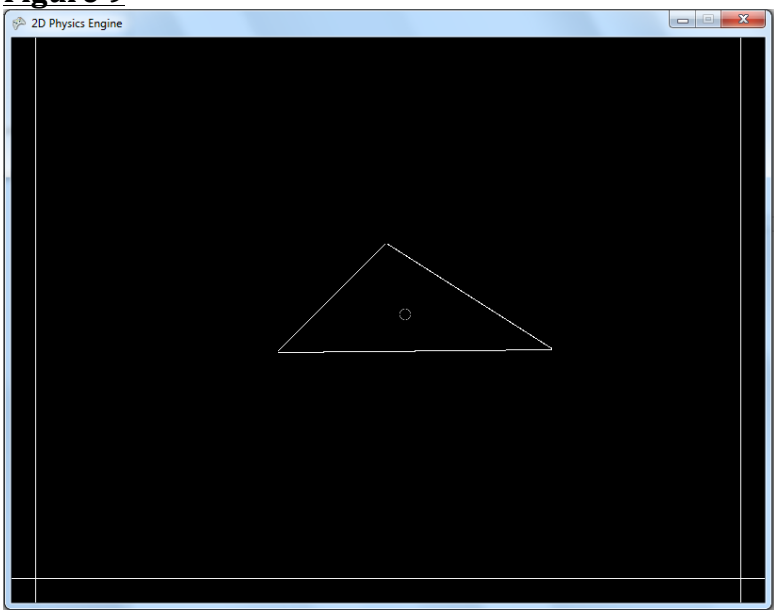


A non-static, gravity-affected circle was then drawn and allowed to collide with the not gravity-affected circle. They bounced off one another.

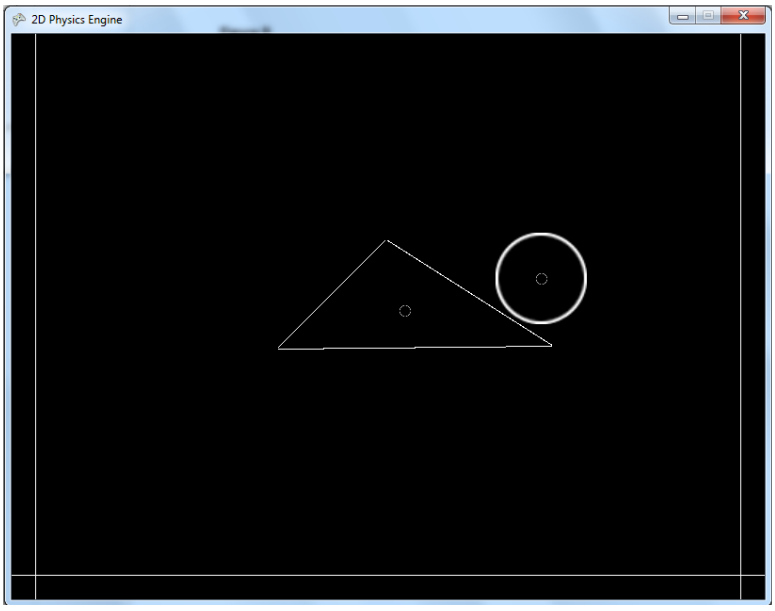


Eventually, the two circles came to rest. The circle that wasn't affected by gravity came to rest in the air.

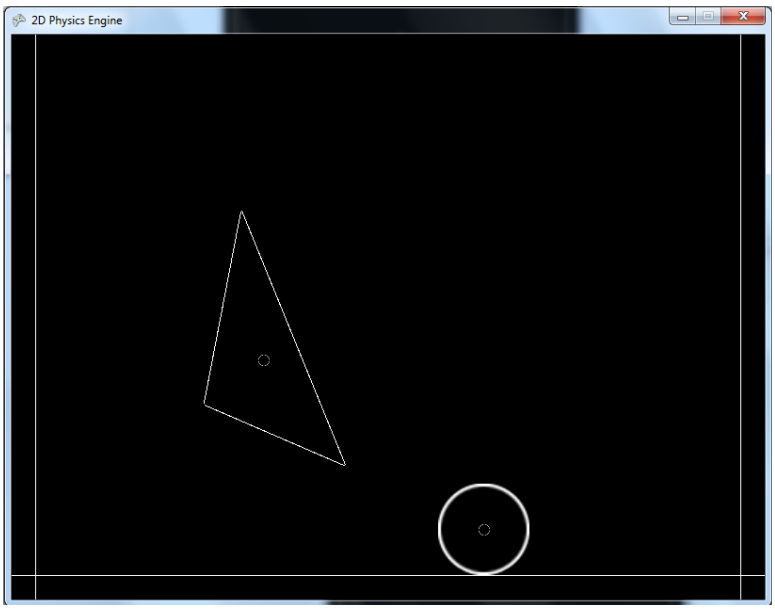
Figure 9



A non-static, not gravity-affected polygon was drawn.

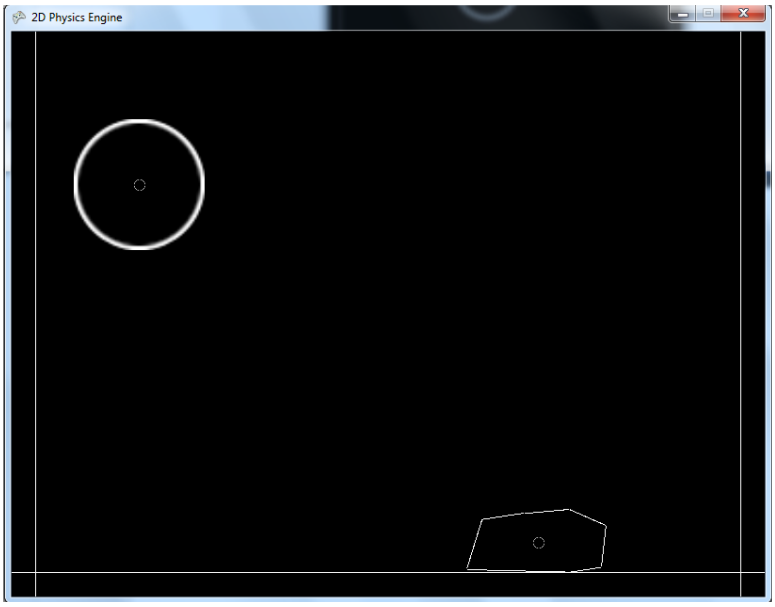


A non-static, gravity-affected circle was then drawn and allowed to collide with the polygon. They bounced off one another and the polygon rotated.

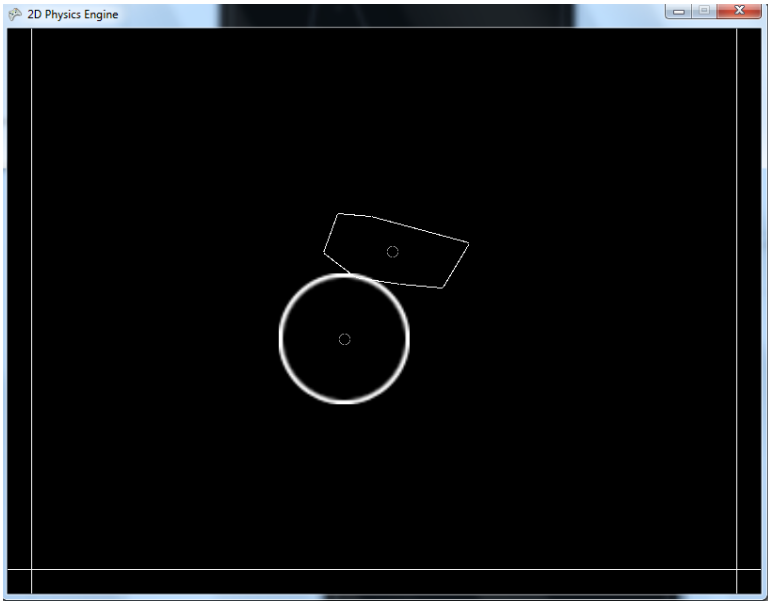


Eventually, the polygon and the circle came to rest. The polygon came to rest in the air.

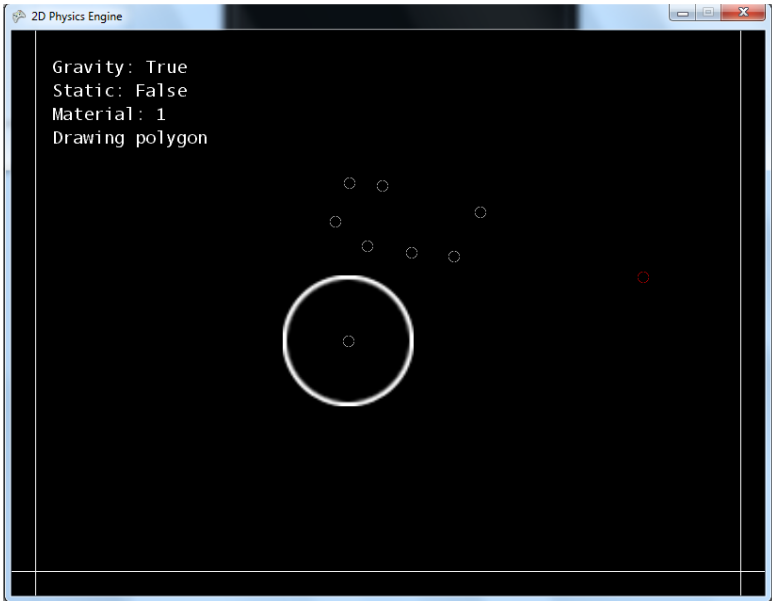
Figure 10



A non-static, not gravity-affected circle was drawn. Followed by a non-static, gravity-affected polygon.

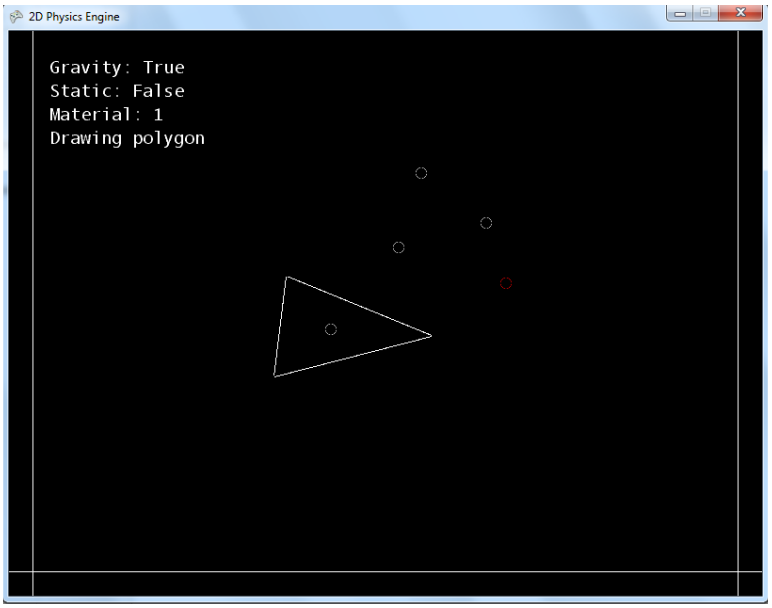


They were then allowed to collide.

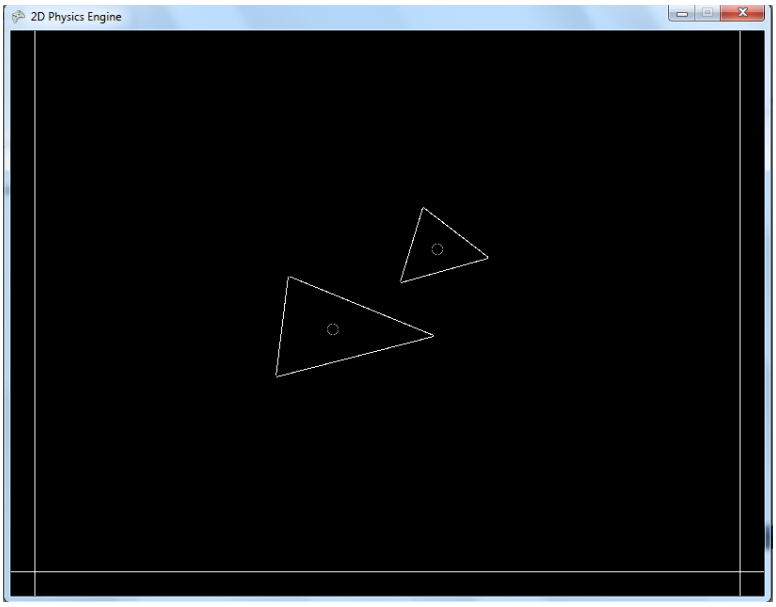


They bounced off one another and the polygon rotated due to the force. Eventually, the circle came to rest in the air and the polygon came to its oscillatory state.

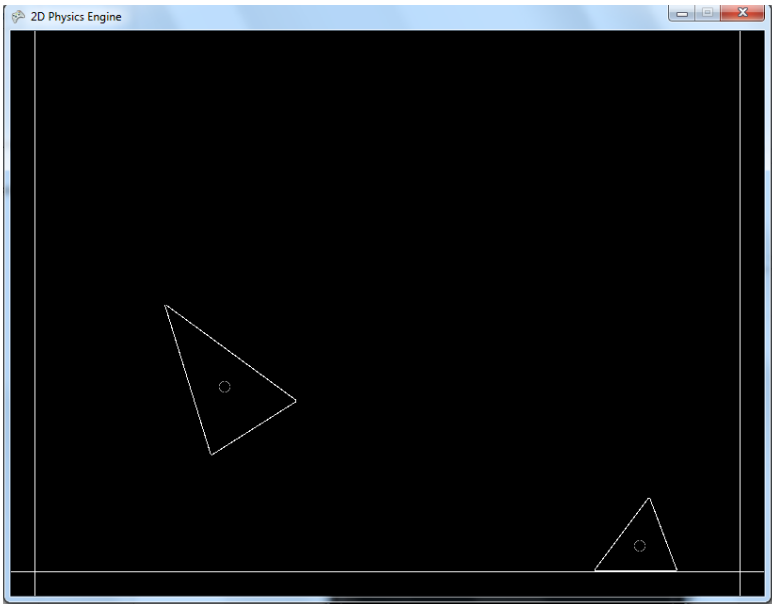
Figure 11



A non-static, not gravity-affected polygon was drawn. Followed by a non-static, gravity-affected polygon.

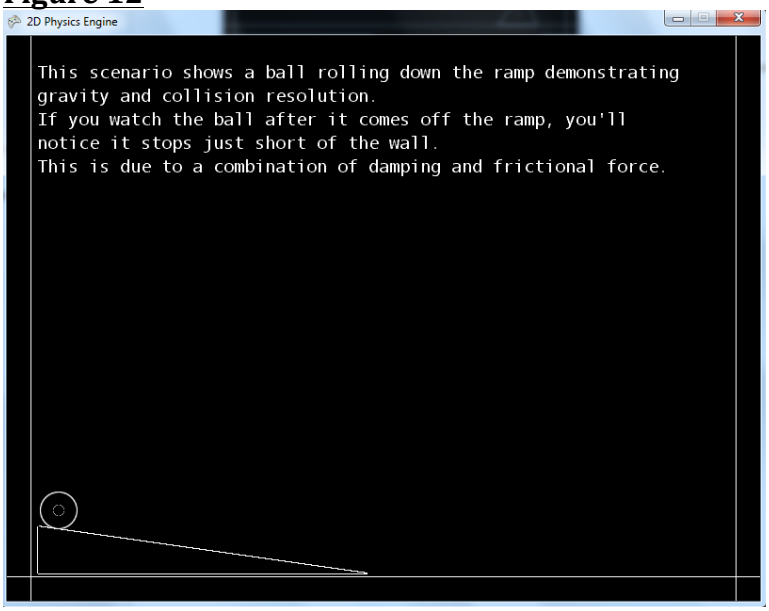


They were allowed to collide and bounced off one another.

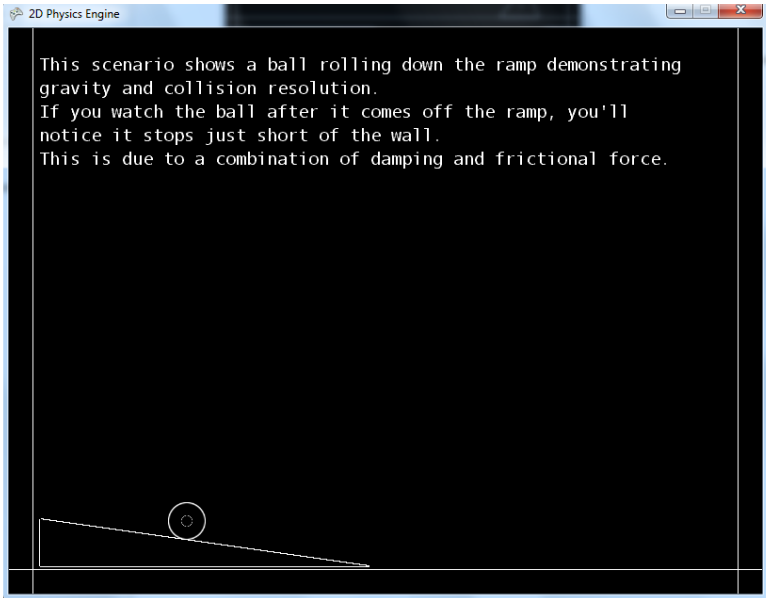


The not gravity-affected polygon eventually came to rest and the gravity-affected polygon came to its oscillatory state.

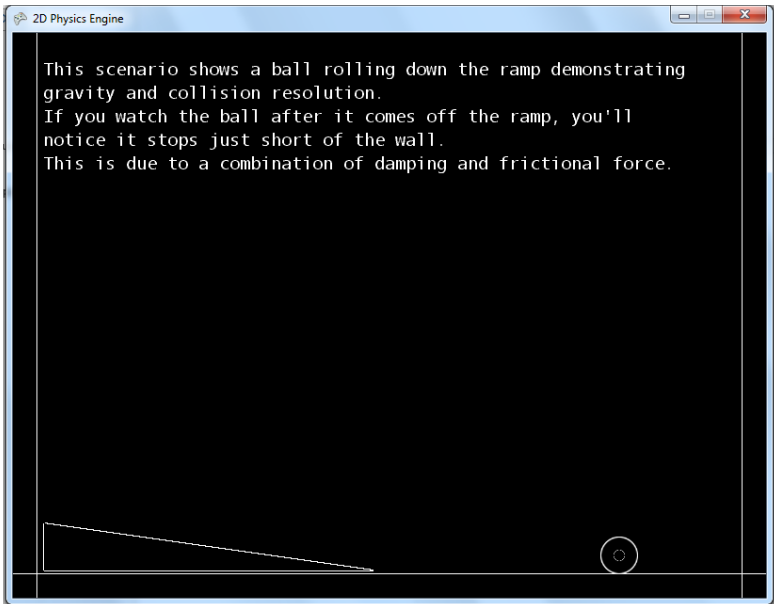
Figure 12



I pressed 1 and the 1st scenario was displayed.

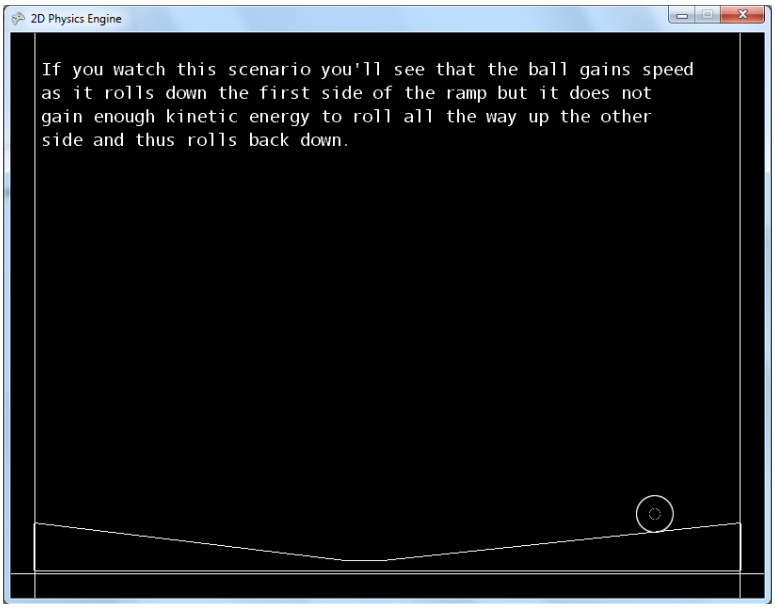


The ball rolled down the ramp and along the floor.

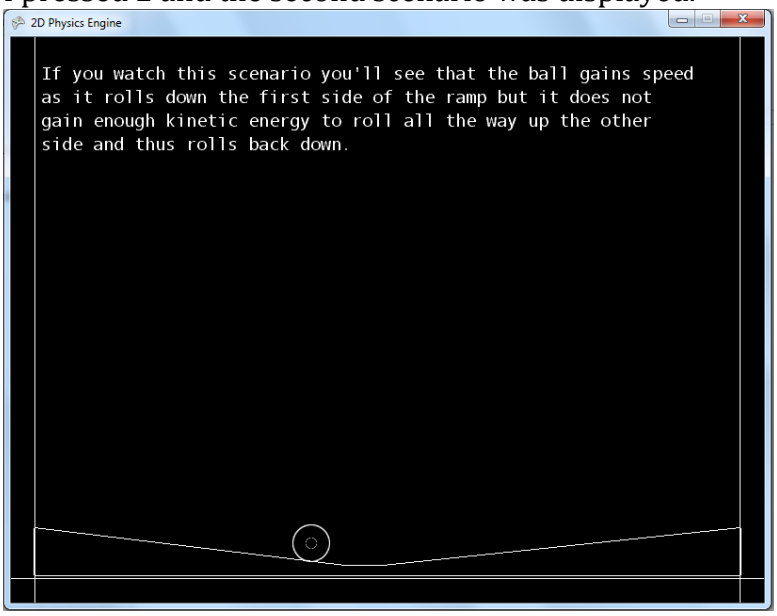


And due to a combination of frictional forces and damping, the ball came to rest just before the wall.

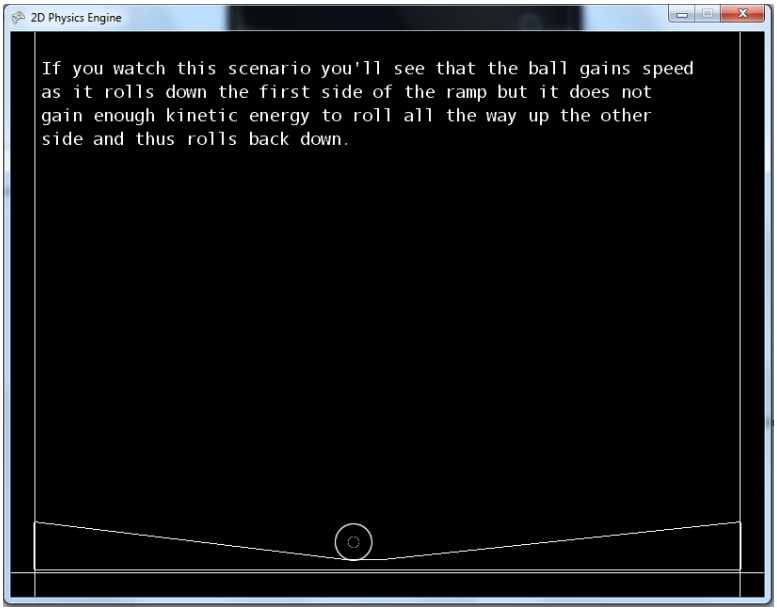
Figure 13



I pressed 2 and the second scenario was displayed.

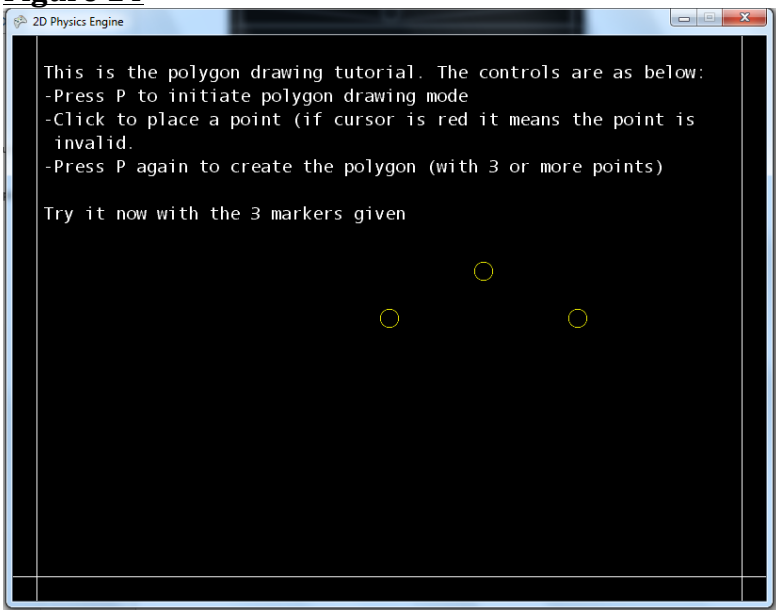


The ball rolled down one side of the shape and began to roll up the other but was stopped due to a lack of energy.



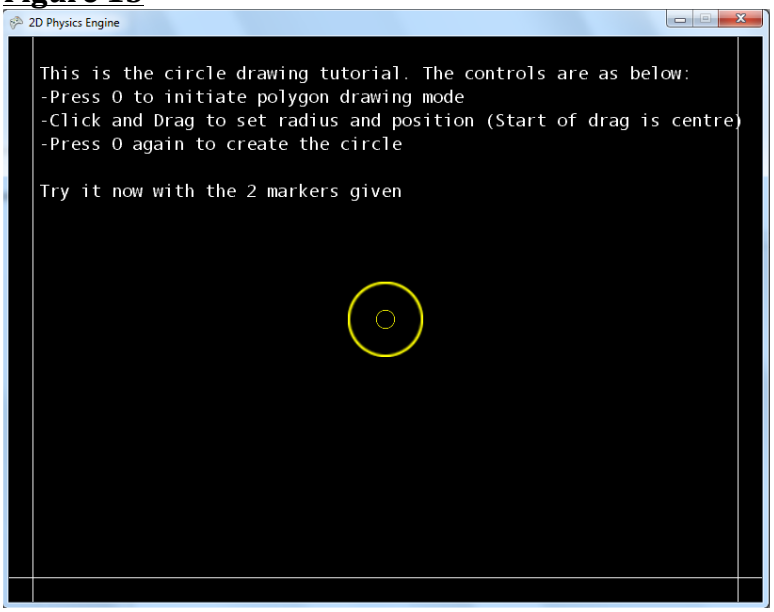
After going back and forth for a while, the ball came to rest in the centre.

Figure 14



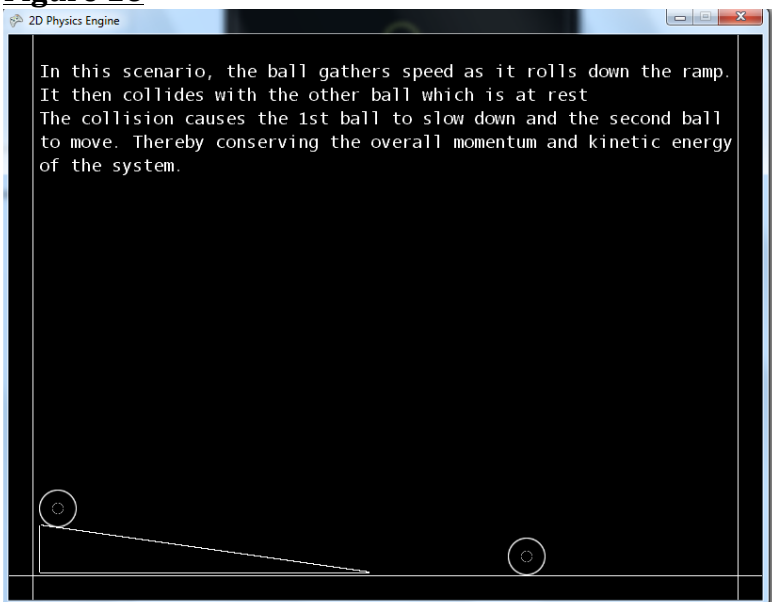
I pressed 3 and the 3rd scenario was displayed.

Figure 15

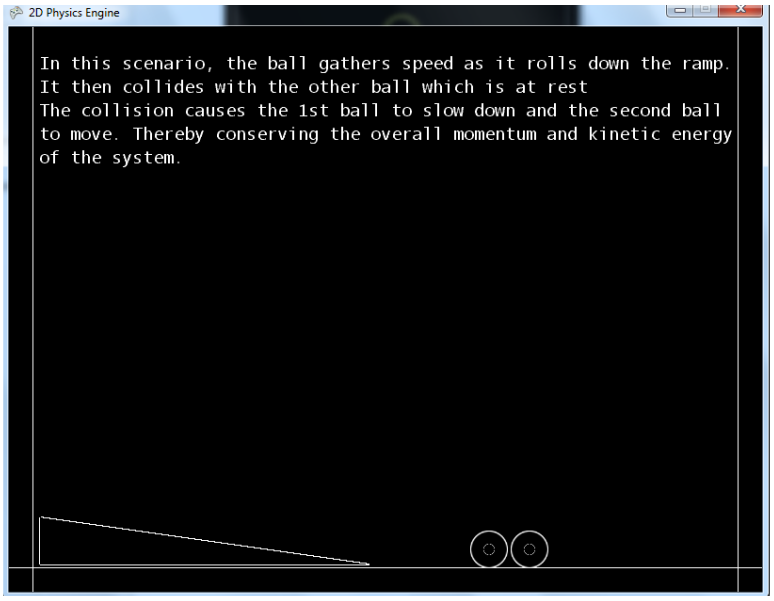


I pressed 4 and the 4th scenario was displayed.

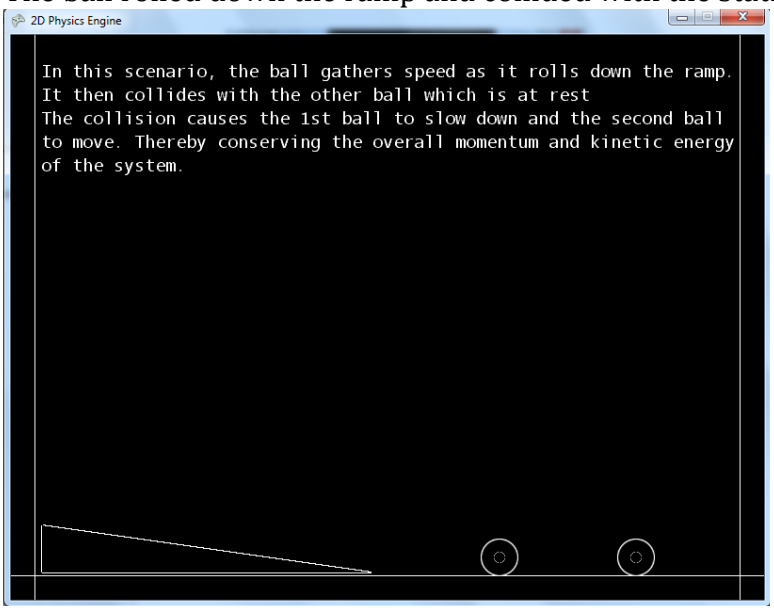
Figure 16



I pressed 5 and the 5th scenario was displayed.

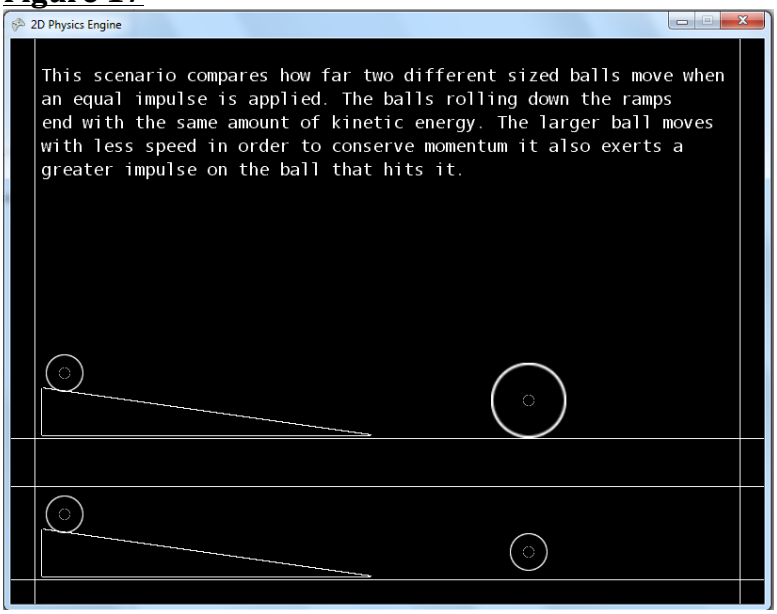


The ball rolled down the ramp and collided with the stationary ball.

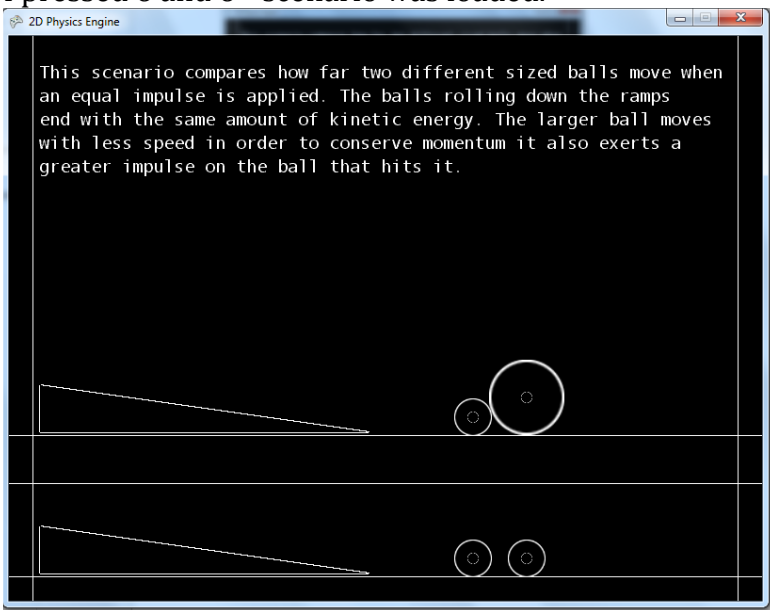


Eventually the two balls came to rest.

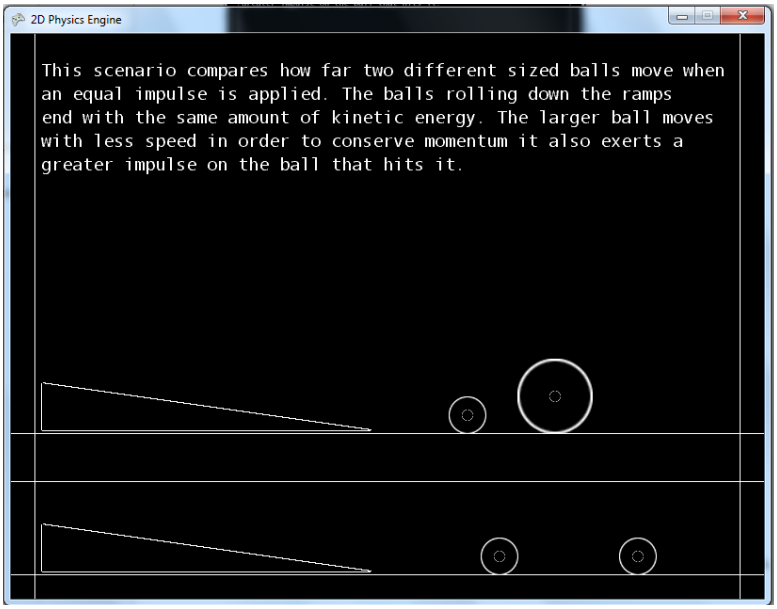
Figure 17



I pressed 6 and 6th scenario was loaded.

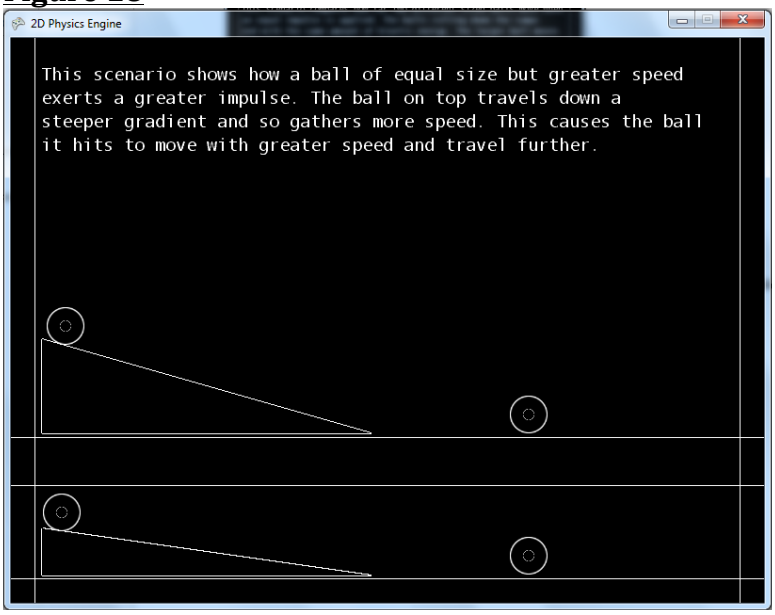


The balls rolled down the ramps and collided with the two different sized stationary balls.

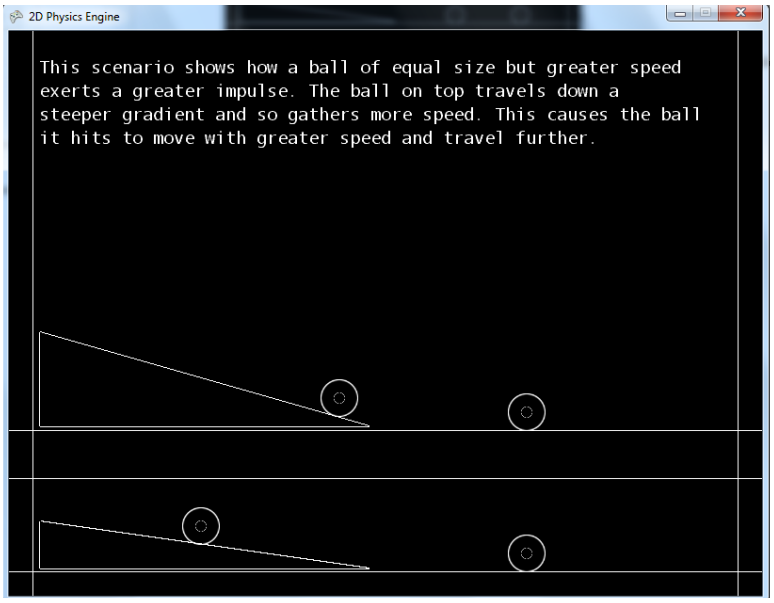


The objects eventually came to rest, the larger ball had rolled less distance than the smaller ball.

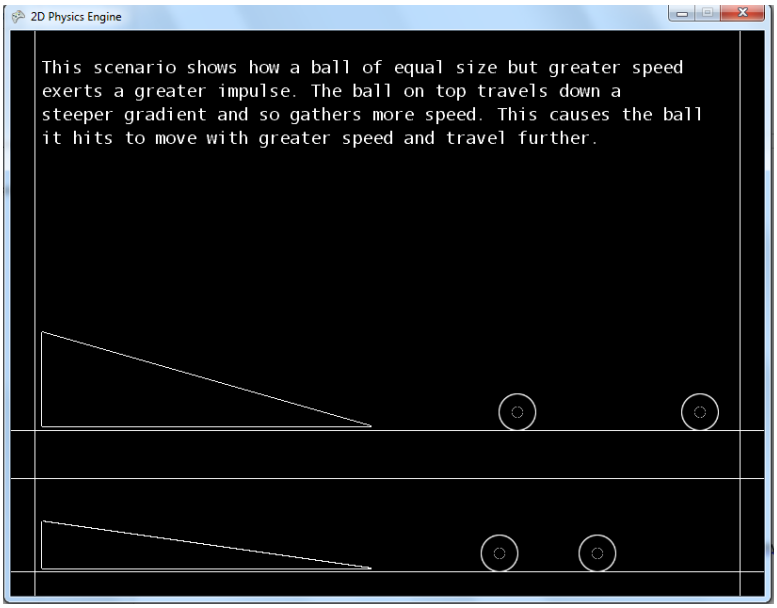
Figure 18



I pressed 7 and the 7th scenario was loaded.

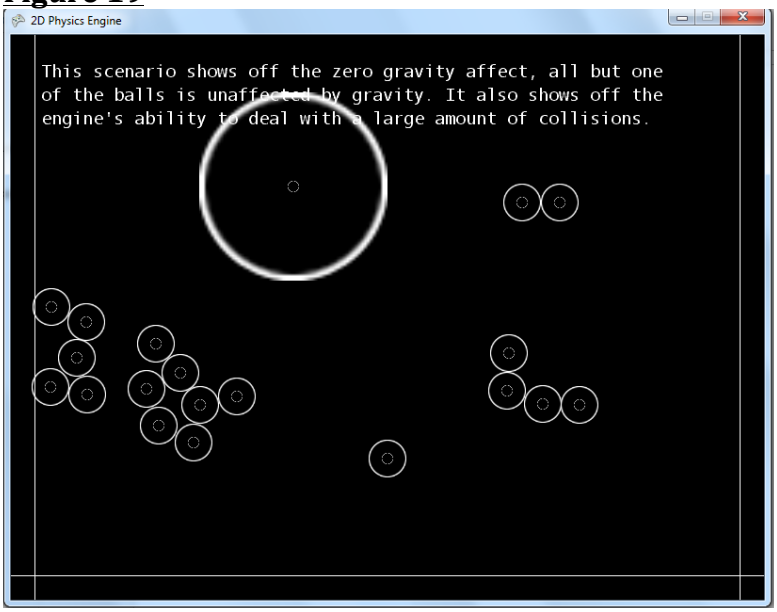


The balls rolled down the ramps, the top ball went faster due to the steeper gradient, they collided with the stationary balls.

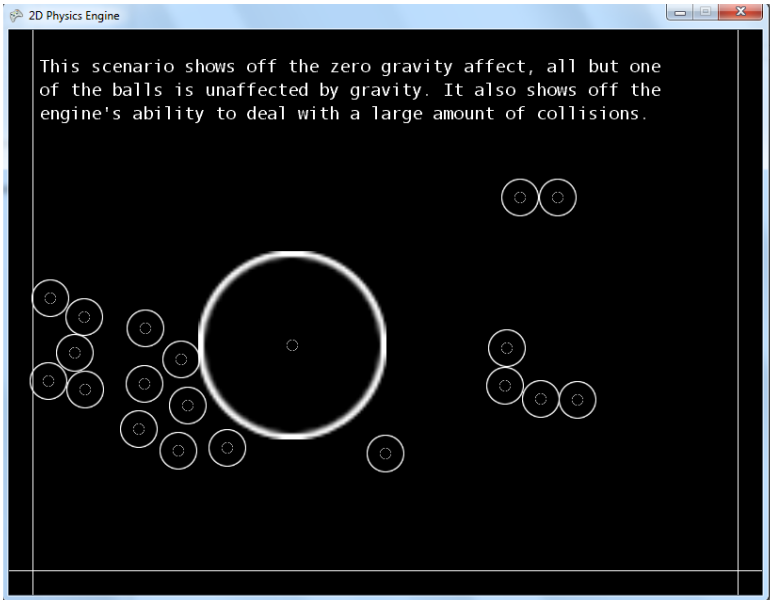


The top ball rolled further as it was hit with a faster projectile.

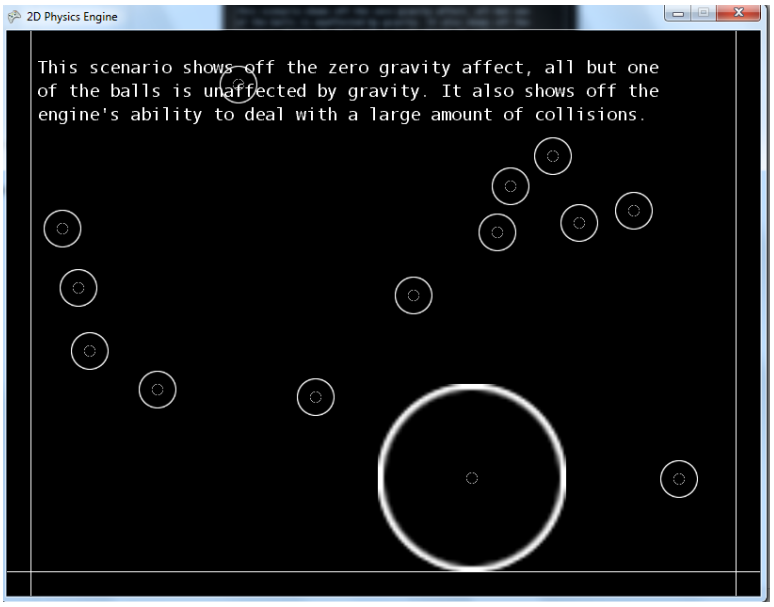
Figure 19



I pressed 8 and the 8th scenario was loaded.

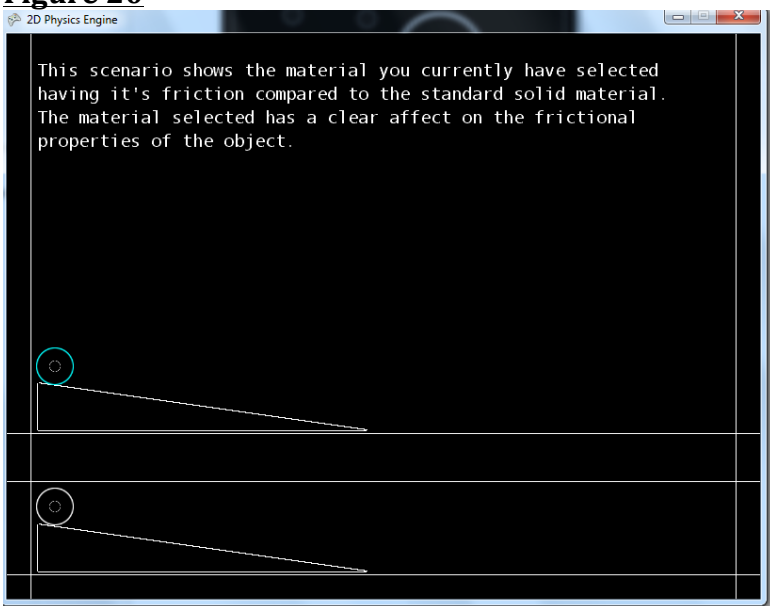


The large ball collided with the smaller circles.

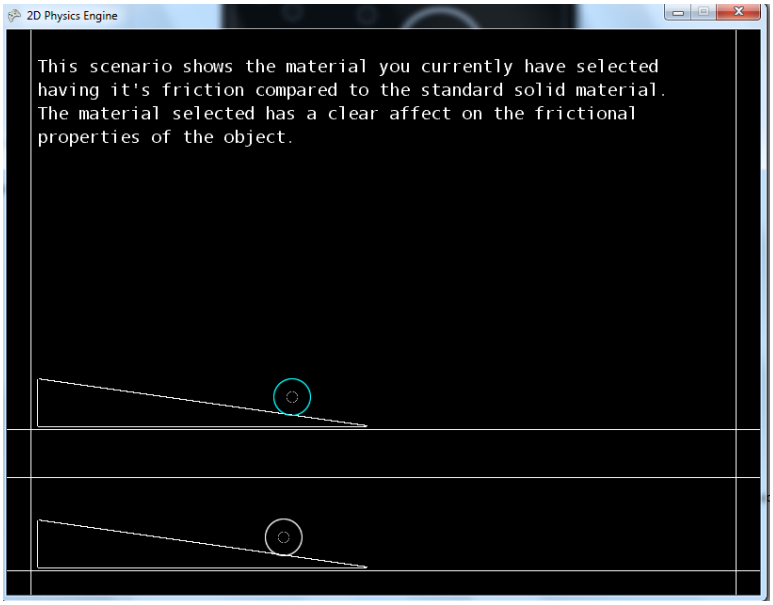


After a series of collisions, all the circles eventually came to rest.

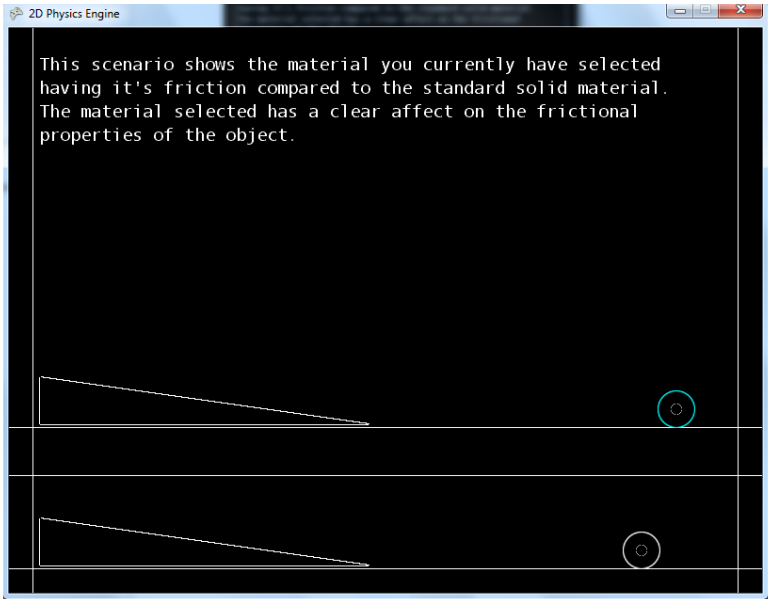
Figure 20



I pressed NUMPAD2 and then pressed 9, the 9th scenario was loaded with the top ball made of material 2 (ice).

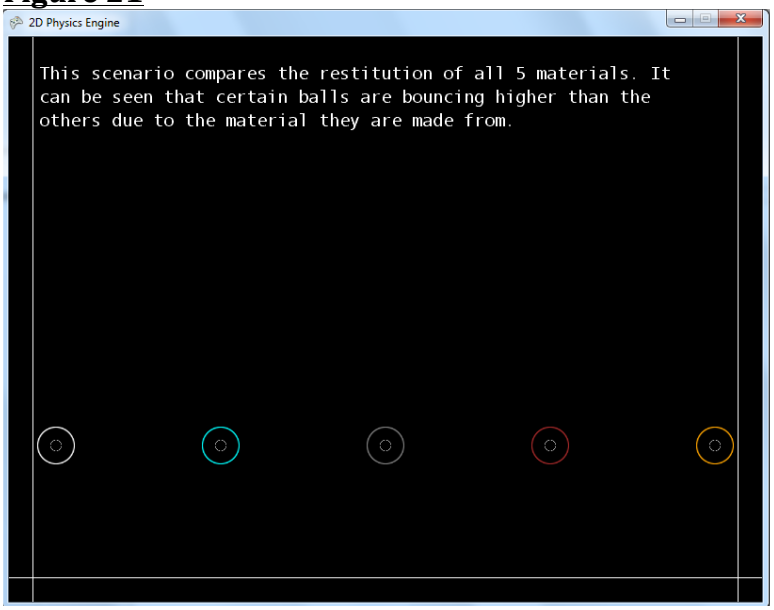


The balls rolled down the ramp.

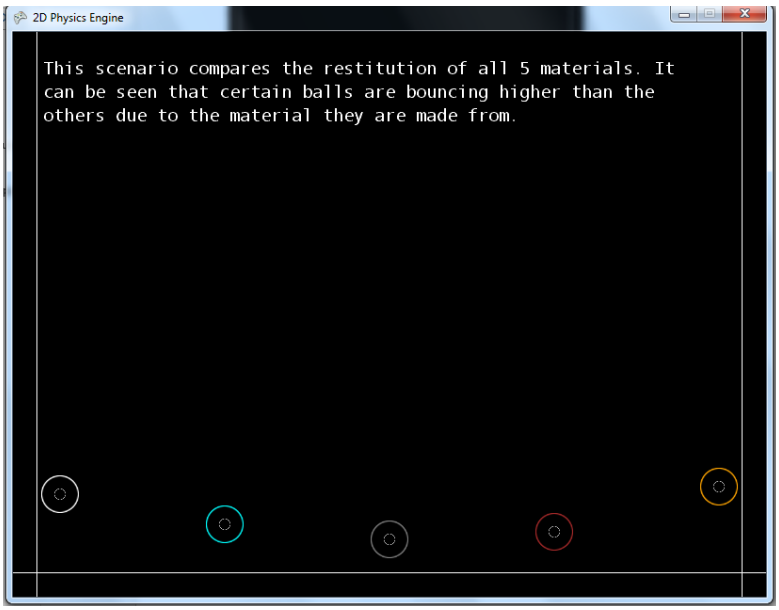


Due to the different frictional properties of the materials, they came to rest at different distances.

Figure 21

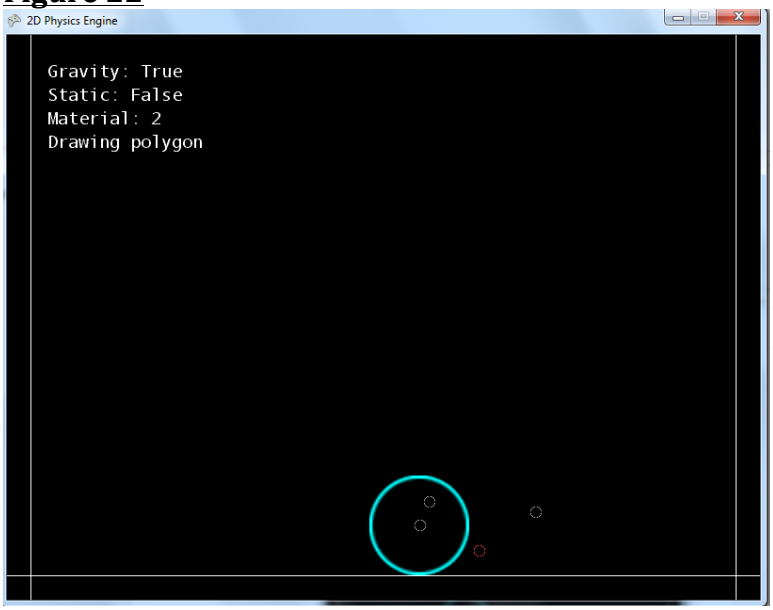


I pressed F1 and the 10th scenario was loaded.

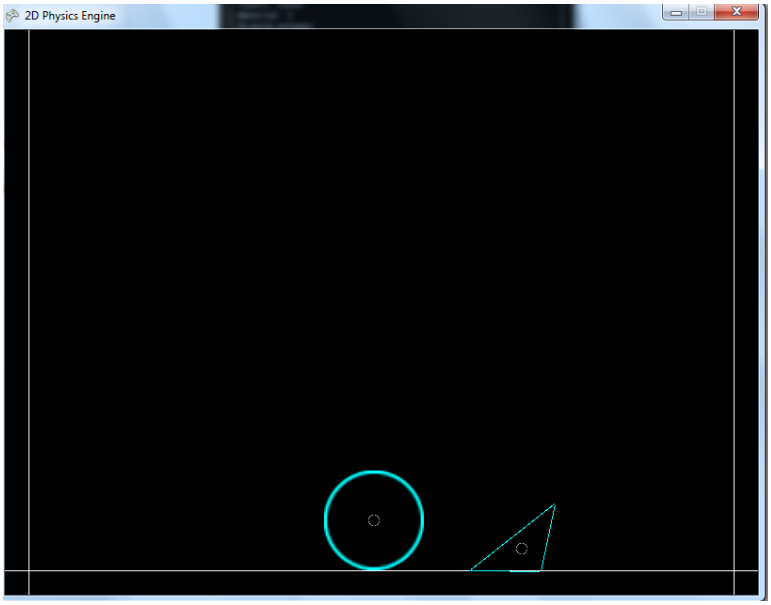


The balls bounced off the floor, they bounced to different heights due to different restitution properties of the materials.

Figure 22



I drew a circle and a polygon which encroached on the circle's space.



The polygon was forced from the circles space and then behaved as normal.

Evaluation

The project was to create a physics engine which met these main areas:

- Gravity (mass causes object to fall)
- Collision Detection
- Collision Resolution
- Rotation
- Friction
- Restitution

All of which are exhibited within the project, as evidenced by testing. There were also extra features added in such as the ability to have multiple materials, toggle whether certain objects feel gravity and allowing the user to manipulate the overall gravity within the system. As such, the solution has been very effective at simulating physics on rigid bodies in a two dimensional space.

Comparison with Objectives

A series of objectives were laid out before beginning the project. The table below shows how the solution stacks up against said objectives.

Objective	Objective Met (Y/N)	Comment
Should be able to detect collision between objects, which is done by representing objects as points and checking if the points overlap.	Y	Collisions between different types of objects are detected. The method outlined in the objective was used to represent polygons and circles were represented as radii.
Every object will have some base properties, which could differ slightly from object to object. So I'll need a way to organise the data.	Y	Each object is its own instance of PhysicsBody which has a corresponding PhysicsShape. Material properties are stored in a static class. All properties of an object are set or calculated upon instantiation. Properties include mass, frictional coefficient and coefficient of restitution.
Should be able to resolve collisions, which I plan to do via impulse resolution.	Y	Collisions between different types of objects behave as expected and are calculated via impulse resolution.
As a part of collision resolution I will have to apply friction to objects when they're rubbing	Y	Objects do experience friction. Each object has two frictional coefficients, one for when

against one another. Which is done via finding a coefficient of friction and a tangent vector and using some formulae from classical mechanics.		the object is static and one when the object is dynamic. The appropriate coefficient is chosen when calculating the frictional force.
I will also need to know which types of objects are colliding so I know which collision method to call.	Y	The project can distinguish between different types of collision. This is done by invoking different methods based on the shape property of two colliding objects.
The objects need to be able to move around with a linear velocity and acceleration	Y	Objects can move around with linear velocity and acceleration. This is done by having properties for velocity and acceleration and having them updated by forces and other factors based on time.
The scene needs to be rendered using a basic pipeline	Y	The scene is rendered properly. A rendering pipeline is created via a queue. Render objects are enqueued and then dequeued when they've been rendered.
A variety of scenes are needed in order to test each aspect, so in order to avoid creating many different set scenes, I will incorporate the ability to draw polygons and circles.	Y	The polygon and circle drawing tools function as expected and don't allow any invalid shapes. In addition to this I designed and incorporated some pre-set scenarios to showcase the engine.

Appraisal

After my own testing, I distributed the engine to some users and gathered some feedback. I posed 3 questions:

1. What did you do when you used the engine?
2. Did you come across any problems with the engine?
3. Any changes you would make?

Responses to Question 1

- I quickly learnt how to use the engine and draw shapes and then ran the scenarios to see what could be done with the engine. After this I created a marble run like structure where I placed a circle near the top of the screen and used static objects to control is motion down to the bottom of the screen. It worked exactly as I intended it to – making me feel the physics behind the engine was very realistic and intuitive.
- I tried out the different capabilities of the software after looking through the examples, which were very useful to understand the capabilities of it. I found the different materials particularly interesting.

Responses to Question 2

- None – everything seemed to work as I would have expected and there were no problems. I also thought the engine was very easy to use despite being completely new to the concept.
- Some objects disappeared when they were forced into too much space, which broke some of the realism.

Responses to Question 3

- Perhaps being able to draw the shapes both clockwise and anticlockwise would be a good improvement, simply to make it even easier to use the engine.

  Aside from this only extension type changes such as more materials and more standard shapes e.g. having a key that draws an equilateral triangle, or being able to re-use already drawn shapes by having them in like a library.
- I would love to see even more options for materials or perhaps other interesting objects such as airflow/fans to demonstrate the capabilities of your engine.

Possible Improvements

Whilst the system has met the set of objectives I laid out at the start of the project and added some extra features to the project, there are still some improvements that would be possible in my mind.

Some ideas that could potentially be covered are Newton's Law of Gravitation, which would mean every object would exert a gravitational force on every other object. I could also have added the ability for the user to manipulate the speed of time. If I combined these two features then it would be possible to set up scenarios that show orbital systems like our solar system.

The other main idea that crossed my mind was fluids, the ability to have water or other liquids would be very useful. The main reason that this wasn't covered is that accurate fluid dynamics is very difficult mathematically (using the Navier-Stokes equation which is a differential equation). Another possibility would be allowing connectors (i.e. strings and rods), which would connect bodies together allowing composite bodies. This would've involved non-rigid bodies in the form

of strings. Also, the strings would have to be able to ignore certain collision functions.
There are also the suggestions from the users who tried out the engine. As one of them said, objects do disappear when forced into a very small space. This is due to the fact that the objects need to be forced apart. Currently the object is forced out in the direction a force would act on it if it were a collision; this means that the shape could easily be forced into another shape and then forced out of that one and off the screen. There are two ways to potentially fix this:

1) Check around the shapes for empty space and force the objects apart so that they both occupy free space.
2) Monitor the amount of free space left on screen and only allow a shape to be drawn if that amount of space is still available (Although this method will still require additional checks.

The same user suggested perhaps including airflow into the engine; this would involve delving into fluid dynamics, as aerodynamics is a sub-field of this (although this is mainly for a truly accurate simulation). I could always have airflow from fans create an area which applies a force on the object and have the amount of force it applies on an object diminish over distance.
Another user suggested allowing shapes to be drawn clockwise and anticlockwise. This would involve expanding on the checks that are made when placing points in the polygon drawing mode. The main difference would be using the other perpendicular to check validity as well as the current checks. The same user also suggested a way of drawing standard shapes, which would be a case of having the engine automatically place points and create an object centred around the mouse's position, or being able to save shapes that have been drawn into files and then be able to load them, this would involve storing the vertices of the shape into a file and then accessing them to draw the shapes automatically.

Summary

As discussed at the beginning of the evaluation, my solution meets my objectives and is very robust. On top of which, extra features have been included. In my testing, I found no issues and the solution deals with erroneous data well. User testing has indicated that there are very few issues with the engine and that it can be used to set up a wide variety of scenarios. There are very few improvements that will improve the robustness of the solution. Most improvements on the current system would just be including more additional features. As such, the project is successful in my eyes.

Appendix

Bibliography

- Box2d, https://en.wikipedia.org/wiki/Box2D, 09:33 11/11/2016
- Nape, http://napephys.com/, 09:40 11/11/2016
- Matter.js, http://brm.io/matter-js/, 9:47 11/11/2016
- Newton's Laws of Motion, https://en.wikipedia.org/wiki/Newton%27s_laws_of_motion#Newton.27s_first_law, 15:07 15/11/2016
- Friction, https://gamedevelopment.tutsplus.com/tutorials/how-to-create-a-custom-2d-physics-engine-friction-scene-and-jump-table--gamedev-7756, 15:15 15/11/2016
- Centre Points, https://en.wikipedia.org/wiki/Centroid#Of_a_finite_set_of_points, 13:57 21/11/2016
- Collision, https://gamedevelopment.tutsplus.com/tutorials/how-to-create-a-custom-2d-physics-engine-the-basics-and-impulse-resolution--gamedev-6331, 14:05 21/11/2016
- Polygon Inertia, https://mathoverflow.net/questions/73556/calculating-moment-of-inertia-in-2d-planar-polygon, 13:05 06/01/2017