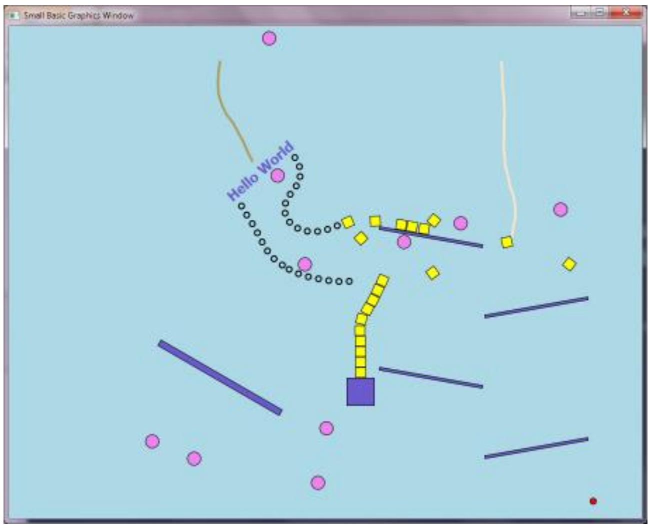
Guide for Small Basic Physics Extension



Summary Overview

This extension uses Box2D (http://box2d.org) as an engine and provides an interface between it and the graphics capabilities of Small Basic.

An older C# port of Box2D and not the current main feature release (C++) version was used for various technical compatibility reasons, namely earlier versions of Small Basic that targeted .Net 3.5 SP1 and 'AnyCPU' platform. However, the port being used for this extension behaves well, with only small modifications.

Shapes and Bodies

The engine calculates the interaction collisions of bodies (or Small Basic shapes) in the presence of gravity. The bodies considered are circles or polygons (rectangles and triangles) and may correspond to any of the Small Basic shapes with these underlying types.

Small Basic Shape	Corresponding Physics Engine Body
Shapes.AddRectangle Shapes.AddText Shapes.AddImage †	Polygon (4 points)
Shapes.AddEllipse	Circle
Shapes.AddTriangle	Polygon (3 points)

† Image shapes can be treated as circles by setting the property **LoadImagesAsCircles** to "True" before loading the images. Also, the **Controls** objects **TextBox** and **Button** can be used, and are treated as rectangles.

Additionally, the FC (or other) extension shapes and controls can be used and are treated by default as rectangles. An exception is the **FCControls.AddPolygon** or **LDShapes.AddPolygon** which creates a polygon that is also supported by this extension as long as the polygon is convex.

There is no support for lines; use a thin rectangle.

The boundary of any shape is that before any zoom transformation using **Shapes.Zoom** is made, so it is best not to use the Small Basic zoom feature in conjunction with this extension.

In addition to these basic shape types, ropes and chains have been implemented for the extension; these are merely linked rectangles or circles that can rotate about each other, but maintain a fixed distance between each link in the rope or chain.

Any shape within the physics engine can either be fixed or moving. Both interact with moving objects, but the fixed ones don't move (i.e. they don't see gravity or other forces and effectively have infinite mass).

When a Small Basic shape is added to the physics engine, its initial position (centre of mass) within the GraphicsWindow is taken to be its current location. Therefore, to create a shape and add it to the physics engine you must follow these steps:

- Create the shape using one of the Small Basic **Shapes.Add…** methods.
- Position the shape using **Shapes.Move**.
- Add the shape to the physics engine using one of the **LDPhysics.Add…** methods.

Alternatively, the following procedure may be used:

- Create the shape using one of the Small Basic **Shapes.Add…** methods.
- Add the shape to the physics engine using one of the **LDPhysics.Add…** methods.
- Position the shape centre using **LDPhysics.SetPosition**.

Physical Properties

Gravity is set to be vertically down on the screen.

Solid borders, where all shapes will bounce, are initially set on all sides of the **GraphicsWindow**. These can be altered using the **LDPhysics.SetBoundaries** method.

Any shape or body in the engine will have numerous properties; here is a list of the main ones that this extension can interact with.

Body Property	Comment
Time	The time interval used is the second, the default time-step is 0.025 or 40th of a second
Pixel	Equivalent to 0.1m
Position	The centre of the body in Small Basic coordinates (pixels).
Angle	The angle of rotation (degrees)
Velocity	Linear velocity at the body centre (pixel/s)
Rotation	The rotation speed of the body (degrees/s)
Impulse	An instantaneous kick that may be applied to a body (mass.pixel/s)
Force	A force (mass times acceleration) applied to a body (mass.pixel/s^2)
Torque	A rotational force (inertia * rotational acceleration) that may be applied to a body (mass.$pixel^2/s^2$)
Friction	Slowing of the body when it rolls or slides over another body (0 to 1 or larger)
Restitution	The bounciness of the body when it collides with another (0 to 1)
Density	The density of the body (default 1 good for most cases) (kg/m^2 or 0.01kg/$pixel^2$)

Mass	The mass of a body (kg)
Inertia	Resistance to rotation depends on shape and mass of body (mass.pixel^2)

Attaching and Grouping

Bodies can be attached to each other using the **AttachShapes** method, for example a block attached to the bottom of a rope, or any shape to another.

When two bodies are attached, their relative separation and rotation are both restricted and they appear to move as one object. It is also possible to attach objects maintaining their relative separation, but allow rotation of the objects using the method **AttachShapesWithRotation**, thus allowing the bodies to rotate (for example a wheel).

Shapes can also be permanently 'glued' together to make compound bodies using the method **GroupShapes**.

The position of a shape (**SetPosition** and **GetPosition**) is its centre of mass for all shapes unless they are connected with the **GroupShapes** method; in which case the reported position is that of the shape to which others are grouped (second argument of the **GroupShapes** method), while the centre of rotation is the centre of mass of the compound body.

It can sometimes be useful to create transparent shapes that are used in a structure, but you don't want to see; in this case just use the Small Basic method **Shapes.SetOpacity**.

To attach one end of a rope or chain to another body, the idea of anchors has been added. These are small transparent bodies, which may be either fixed or moving like any other body.

Note that the first shape is added to the second. In the case of **GroupShapes** the new compound shape is the second shape and in the case of the **AttachShapes** methods the main body is the second, with the first being added to it. For example think of grouping a car door to the car or attaching a wheel to the car, not the car to the wheel. This will help considerably with overall stability of compound structures.

The Game Loop

Once the bodies are defined, the physics engine will update their position, rotation etc. at each time-step. The physics extension also updates the graphical display of the shapes as they move and rotate in the Small Basic GraphicsWindow. A time-step is called using the method **LDPhysics.DoTimestep**. This will generally be repeatedly called in a loop (called the 'main game loop').

```
While ("True")
  LDPhysics.DoTimestep()
  Program.Delay(20)
EndWhile
```

The 'game loop' concept is common to all dynamic interaction type software and worth using from the start.

Other game-play interactions are made inside this game loop, such as moving barriers, firing bullets, adding or removing bodies etc.

Beyond Small Basic

Use the examples listed below – they are more than just a random set of things that can be done; but demonstrate important features and concepts. There is also lots of web documentation on 2D Physics, simulation and Box2D in particular if you fancy using it directly, or just understanding how to build models since the Small Basic extension interface pretty much follows what Box2D can do.

There are loads of small web based physics programs, many using Box2D ports to JAVA or Flash, which could be the next step once you have the concepts of setting up and interacting with the physics engine.

Additional Comments

Only shapes that are connected to the physics engine take part in the motion physics, for example you may add normal shapes (e.g. a gun and not connect it to the physics engine). Once a shape is connected to the engine, it is best to only interact with it through the methods provided by the extension. All positions are in the Small Basic **GraphicsWindow** pixels and refer to shape centres.

Tunnelling and Stability

One issue that Box2D has difficulty with is small fast moving objects that can 'tunnel' through other shapes without being deflected (see the **SetBullet** option).

Another problem is shapes of very different size and hence mass, especially large shapes when they are connected together. It may be necessary to reduce the density of large bodies or increase the density of small bodies. Generally the default density of 1 is good. Resist the temptation to connect too many shapes together.

The author of Box2D (Erin Catto) states the limitations very clearly:

- Stacking heavy bodies on top of much lighter bodies is not stable. Stability degrades as the mass ratio passes 10:1.
- Chains of bodies connected by joints may stretch if a lighter body is supporting a heavier body.
    - For example, a wrecking ball connected to a chain of light weight bodies may not be stable.
    - Stability degrades as the mass ratio passes 10:1.
- There is typically around 0.5cm of slop in shape versus shape collision.
- Continuous collision does not handle joints. So you may see joint stretching on fast moving objects.

It may be possible to improve the stability of some 'difficult' models using the **TimestepControl** settings, but the defaults look good for most cases. It is often better to consider reducing mass and size variation between shapes. Again, Erin states:

> “Box2D uses a computational algorithm called an integrator. Integrators simulate the physics equations at discrete points of time. This goes along with the traditional game loop where we essentially have a flip book of movement on the screen. So we need to pick a time step for Box2D. Generally physics engines for games like a time step at least as fast as 60Hz or 1/60 seconds. You can get away with larger time steps, but you will have to be more careful about setting up the definitions for your world. We also don't like the time step to change much. A variable time step produces variable results, which makes it difficult to debug. So don't tie the time step to your frame rate (unless you really, really have to).

> In addition to the integrator, Box2D also uses a larger bit of code called a constraint solver. The constraint solver solves all the constraints in the simulation, one at a time. A single constraint can be solved perfectly. However, when we solve one constraint, we slightly disrupt other constraints. To get a good solution, we need to iterate over all constraints a number of times.
>
> There are two phases in the constraint solver: a velocity phase and a position phase. In the velocity phase the solver computes the impulses necessary for the bodies to move correctly. In the position phase the solver adjusts the positions of the bodies to reduce overlap and joint detachment. Each phase has its own iteration count. In addition, the position phase may exit iterations early if the errors are small."

Events and Teleporting

Do not call the physics methods inside Small Basic event subroutines directly, rather set flags that can be processed in a main game loop, since the event subroutines in Small Basic are performed on separate threads and may try to update variables while the physics engine is in the middle of its calculations, resulting in unpredictable results or a crash.

Debugging what is happening can often be achieved by increasing the delay used inside the 'game loop' to slow the motion, and using **TextWindow.WriteLine**.

If you move an object using **SetPosition**, then the next time **DoTimestep** is called the object will be instantly in its new position (teleported) and interact from there. There will be no interactions with any bodies it 'passes through' or touches before and after its position was manually changed.

The Universe AABB

The engine 'universe' is bounded by an AABB (axis aligned bounding box). Anything inside this region takes part in the physics and anything outside it is considered frozen or dead and takes no part in any calculations.

Initially this region is (-100,200) in both the X and Y directions. Since these are internal units of m, they correspond to (-1000,2000) pixels and represents a region larger than any likely window. Additionally, the edges of the GraphicsWindow are defaulted to be solid so that all shapes bounce on the window sides and cannot penetrate beyond the visible window area.

So why is this even worth mentioning? You may want physics to continue being calculated outside the visible window, for example a scrolling platform type game where the player may move off to the right, but you still want activity on the left that gets scrolled off-screen to continue. To achieve this you will need to remove some boundaries using the **SetBoundaries** method.

Consider the **PanView** option to scroll a view, perhaps to keep a moving player sprite central. All frozen shapes can be removed using the **RemoveFrozen** method. You may also want to increase the size of the AABB using **SetAABB**; this must be done at the beginning, calling a **Reset** immediately afterwards.

Fixture Types

There are several ways a Small Basic shape may be added to the physics engine. The shapes may move or not (anchors), rotate or not or even have some special characteristics. Some of these behaviours may be modified after the shape is added.

Below are a set of tables listing the LDPhysics methods by group with some basic comments.

Fixture Types	Methods	Visible	Interacts	Can Move	Can Rotate
FixedAnchor	AddFixedAnchor		x		
MovingAnchor	AddMovingAnchor		x	x	x
FixedShape	AddFixedShape	x	x		
MovingShape	AddMovingShape	x	x	x	x
InactiveShape	AddInactiveShape	x			

Secondary Fixtures	Methods	Comments
Chain	AddChain RemoveChain ChainColour	Connects a chain between 2 existing shapes
Rope	AddRope RemoveRope RopeColour	Connects a rope between 2 existing shapes
Explosion	AddExplosion	Temporarily add a large number of invisible shapes moving fast away from explosion center

Fixture Modifiers	Methods	Comments
Bullet	SetBullet UnsetBullet	Prevents small fast moving shapes from ‘tunnelling’
Tire	SetTire MoveTire TurnTire BrakeTire GetTireProperties SetTireProperties GetTireInformation	A shape can be controlled in a ‘top-down’ zero gravity environment
Change Type	ToggleMoving ToggleRotation ToggleSensor	Modify a shape to allow moving or rotation

Fixture Removal	Methods	Comments
Remove	RemoveShape	Remove a shape completely
Disconnect	DisconnectShape	Remove a shape from interacting, while leaving it visible
Frozen	RemoveFrozen	Remove shapes outside the interaction region AABB

Fixture Connections	Methods	Comments
Attach	AttachShapes AttachShapesWithRotation DetachShapes	Connect shapes to move together, but allow some independent interaction

Group	GroupShapes UngroupShapes	Connect shapes solidly, as one new shape
Joints	AttachShapesWithJoint SetJointMotor DetachJoint	Connect shapes with a variety of joint types

Fixture Control	Methods	Comments
Get Properties	GetAllShapesAt GetAngle GetCollisions GetContacts GetInertia GetMass GetPosition GetRotation GetShapeAt GetVelocity RayCast	Get shape properties
Setup Properties	SetAngle SetDamping SetGroup SetPosition SetRotation SetVelocity	Set shape properties recommended during setup
Simulation Properties	SetForce SetImpulse SetTorque	Set shape properties recommended during simulation while time stepping
Sleeping	WakeAll	Wake all sleeping shapes

World	Methods	Comments
Creation	SetAABB SetBoundaries Scaling Help ReadJson WriteJson	Set the world geometry and pixel/m scaling
Remove	Reset	Delete everything and return to default settings
Gravity	SetGravity SetShapeGravity	Set the world gravity
Timestep	TimeStep TimestepControl PositionIterations VelocityIterations VelocityThreshold DoTimestep	Control numerical aspects of the time step calculations
Fixtures	LoadImagesAsCircles MaxPolygonVertices MaxProxies	Control aspects of shapes that can be added
Control	PanView FollowShapeX FollowShapeY	Scroll the world and pan with shapes as they move

BoxShape
GetPan

Note on Units

Since this is a physics extension I have made some effort to honour the rigor in Box2D and stick to consistent units so that position, velocity, acceleration and rotation are consistent with the mass, inertia and applied forces and torques in the Small Basic units.

This keeps the physics right, but the units can be ignored and just use values for force and torque etc. that work in the context of your model.

The underlying Box2D units are not appropriate to the pixel units used in Small Basic, so a conversion factor 10 (pixels/m) was applied, while maintaining the Box2D time unit of seconds. 1 pixel corresponds to 0.1m in Box2D internal units. Therefore the unit of acceleration due to gravity (10m/s^2) is 100 pixel/s^2.

To use the Force, Impulse and Torque it is easiest to use the Mass and Moment of Inertia that can be obtained for shapes.

Density is set to be in units of kg/m^2. Thus, a square shape with sides 10 * 10 pixels will have area of 100 pixel^2 and therefore internal area of 1 m^2 and therefore a mass of 1 kg using the default density of 1 kg/m^2.

- Force = Mass x Acceleration
- Impulse = Force x Time
- Torque = Moment of inertia x Angular acceleration (radians/s^2)

Therefore:

```
LDPhysics.SetForce(shape,0,-LDPhysics.GetMass(shape))
LDPhysics.SetImpulse(shape,0,-LDPhysics.GetMass(shape)/LDPhysics.TimeStep)
```

Either of the above commands will accelerate the shape upwards at a rate of 1 pixel/s^2.

```
LDPhysics.SetTorque(shape,LDPhysics.GetInertia(shape))
```

The above command will apply a rotational acceleration to the shape clockwise at a rate of 1 radian/s^2 (about 57 degrees/s^2). I have stuck to radian/s^2, rather than degree/s^2 since there are physical relationships between applied forces, torques and power that I want to maintain.

Installing the Extension

In order to use this extension you must add the files 'LitDev.dll' and 'LitDev.xml' to the lib sub-folder within the Small Basic installation folder. You may need to create the lib folder if it doesn't already exist (if other extensions have not already been installed).

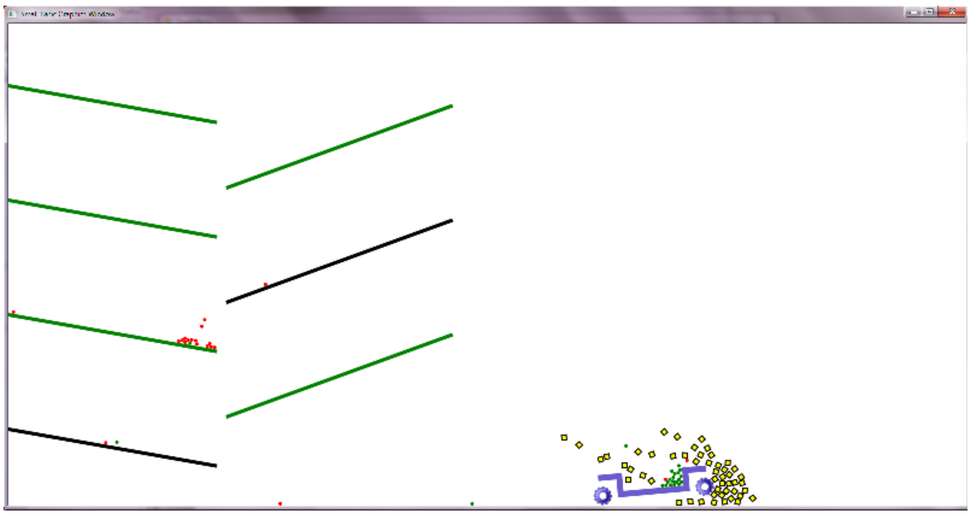
The Small Basic installation directory is usually found at:

• C:\Program Files\Microsoft\Small Basic	(32 bit Windows)
• C:\Program Files (x86)\Microsoft\Small Basic	(64 bit Windows)

Therefore, on a 32 bit Windows the following files will be added:

- C:\Program Files\Microsoft\Small Basic\lib\LitDev.dll
- C:\Program Files\Microsoft\Small Basic\lib\LitDev.xml

The other files in this extension zip include documentation, the examples used in the documentation and some other Small Basic samples. These files are not required for the extension to work and should not be saved in the lib folder, perhaps put them in your Documents folder.



Examples

The following is a set of examples that briefly demonstrates the main features of this extension, with an emphasis on how to get the engine to achieve convincing physical simulations.

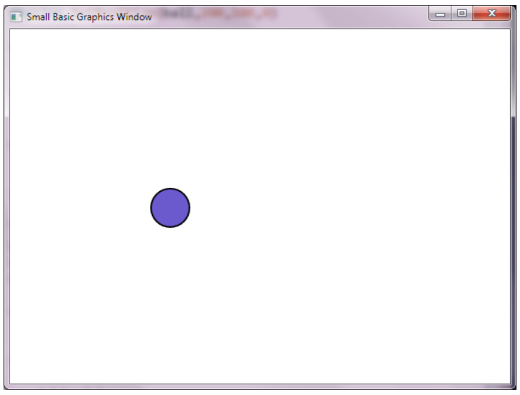
They are just a beginning; the limits depend mainly on imagination and ingenuity, and perhaps performance, when you should be moving beyond Small Basic.

Something that looks convincing may be completely physically wrong; it just looks plausible – great for games, but perhaps not for weather forecasting or engineering simulators. Since this is about simulating physics I add the warning not to believe any simulation (not specifically Box2D but any computer simulation) has any bearing on reality, regardless of how realistic and impressive the graphics look – in fact I would suggest the more effort has been spent making it look visually realistic the less realistic the simulation is likely to be.

The main steps for any dynamic visual (game) program are the same, also for Small Basic and especially using this extension are:

- Setup and initialisation
  - Create all the sprites and game-play variables, perhaps also an intro or instructions.
  - Add the sprites to the physics engine and position them or set initial conditions such as velocity etc.
- Create a main game loop
  - This is usually a **While … EndWhile** loop that repeats while the game plays, taking user input, performing any calculations required and updates the display after all time-step calculations have been performed, hopefully at a decent fps (frame per second).
- Write any subroutines to perform setup or game calculations
  - This helps to isolate calculations from the logic of the main game loop and improve reusability and development of a more complex project.
- Write event subroutines that set flags that are handled in the main game loop or its subroutines
  - In Small Basic, events are handled asynchronously (at the same time on a different thread) with the main program game loop. For this reason if we interact with the physics engine from inside an event subroutine, the program will try to perform our event action at the same time as the main game loop while sharing the same data – this will crash the program.
  - For this reason we must set a flag (just a simple variable) to show that the event occurred (e.g. a mouse click) and change the game play accordingly in the main game loop, where calculations and updates are synchronous (one instruction follows the previous on the same thread).

Example 1 – A falling ball



Create the objects in Small Basic

```
ball = Shapes.AddEllipse(50,50)
```

Attach to the physics engine

Give the ball friction=0 and restitution=1 (a bouncy ball).

```
LDPhysics.AddMovingShape(ball,0,1,1)
```

Set its initial position near the top of the window with zero rotation (irrelivant for a circle) – if we don't set its position in the engine it will take the current position of the shape.

```
LDPhysics.SetPosition(ball,200,100,0)
```

Create a game loop

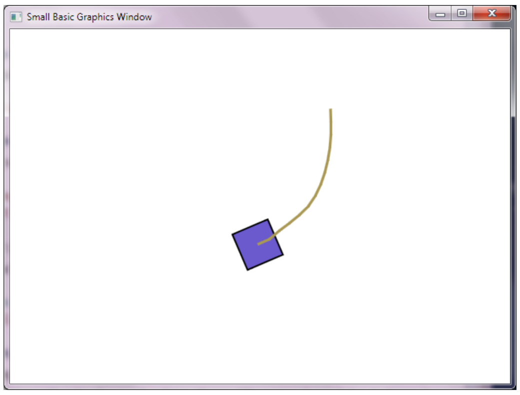
This is the repeating loop, where time-steps are performed to show the motion. The physics engine will update the position and rotation of the bodies and redraw them. We put a short delay to keep it smooth.

```
While ("True")
  LDPhysics.DoTimestep()
  Program.Delay(20)
EndWhile
```

The whole thing

```
ball = Shapes.AddEllipse(50,50)
LDPhysics.AddMovingShape(ball,0,1,1)
LDPhysics.SetPosition(ball,200,100,0)

While ("True")
  LDPhysics.DoTimestep()
  Program.Delay(20)
EndWhile
```

Example 2 – A block hanging on a rope



Create the objects in Small Basic

We only need the block shape, since the rope method is special to the physics engine.

```
block = Shapes.AddRectangle(50,50)
```

Attach to the physics engine

Add the block and set its initial position – note the block is a moving shape.

```
LDPhysics.AddMovingShape(block,0,1,1)
LDPhysics.SetPosition(block,200,100,0)
```

Add a fixed anchor point 200 pixels to the right of the block and attach a rope between the fixed anchor and the block.

```
anchor = LDPhysics.AddFixedAnchor(400,100)
LDPhysics.AddRope(anchor,block)
```

Create a game loop

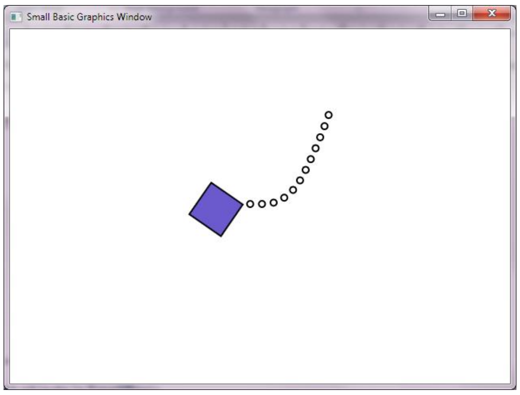
This is the repeating loop, where time-steps are performed to show the motion. The physics engine will update the position of the bodies and redraw them.

```
While ("True")
  LDPhysics.DoTimestep()
  Program.Delay(20)
EndWhile
```

The whole thing

```
block = Shapes.AddRectangle(50,50)
LDPhysics.AddMovingShape(block,0,1,1)
LDPhysics.SetPosition(block,200,100,0)

anchor = LDPhysics.AddFixedAnchor(400,100)
LDPhysics.AddRope(anchor,block)

While ("True")
  LDPhysics.DoTimestep()
  Program.Delay(20)
EndWhile
```

Example 3 – A block hanging on a chain from its corner



To hang from a position that is not at the centre of a shape we need to create an anchor on the corner of the block and attach the anchor to the block and the chain.

Create the objects in Small Basic

We only need the block shape, since the chain method is special to the physics engine.

```
block = Shapes.AddRectangle(50,50)
```

Attach to the physics engine

Add the block and set its initial position – note the block is a moving shape.

```
LDPhysics.AddMovingShape(block,0,1,1)
LDPhysics.SetPosition(block,200,100,0)
```

Add a moving anchor positioning it on one of the corners of the block and then attach it to the block.

```
anchor1 = LDPhysics.AddMovingAnchor(225,125)
LDPhysics.AttachShapes(anchor1,block)
```

Add a fixed anchor point 200 pixels to the right of the block and attach a chain between the fixed anchor and the anchor on the block.

```
anchor2 = LDPhysics.AddFixedAnchor(400,100)
LDPhysics.AddChain(anchor1,anchor2)
```

Create a game loop

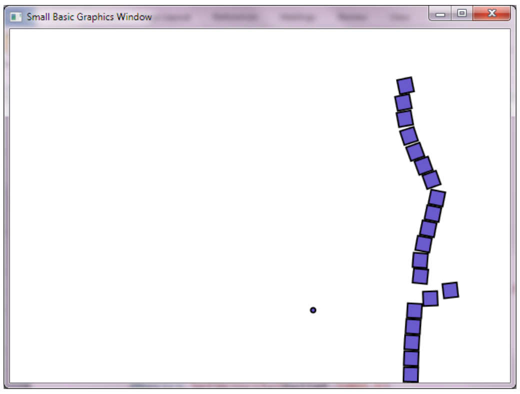
This is just the same as before.

The whole thing

```
block = Shapes.AddRectangle(50,50)
LDPhysics.AddMovingShape(block,0,1,1)
LDPhysics.SetPosition(block,200,100,0)

anchor1 = LDPhysics.AddMovingAnchor(225,125)
LDPhysics.AttachShapes(anchor1,block)

anchor2 = LDPhysics.AddFixedAnchor(400,100)
LDPhysics.AddChain(anchor1,anchor2)

While ("True")
```

```
  LDPhysics.DoTimestep()
  Program.Delay(20)
EndWhile
```

Example 4 – A bullet hitting a pile of blocks



Create the objects in Small Basic and attach to the physics engine

Create 20 blocks and stack them up from the bottom of the window. We don't really need to put the created blocks in an array unless we want to access them individually later.

```
For i = 1 To 20
  block[i] = Shapes.AddRectangle(20,20)
  LDPhysics.AddMovingShape(block[i],0.3,0.8,1)
  LDPhysics.SetPosition(block[i],500,GraphicsWindow.Height-20*i+10,0)
EndFor
```

Create a small bullet and position it at the left of the screen and give it a large positive initial velocity. We could give it more mass using a larger density or set it as a bullet type body using **LDPhysics.SetBullet**, but here it seems OK as it is.

```
bullet = Shapes.AddEllipse(8,8)
LDPhysics.AddMovingShape(bullet,0,1,1)
LDPhysics.SetPosition(bullet,50,200,0)
LDPhysics.SetVelocity(bullet,1000,0)
```

Create a game loop

This is just the same as before.

The whole thing

```
For i = 1 To 20
  block[i] = Shapes.AddRectangle(20,20)
  LDPhysics.AddMovingShape(block[i],0.3,0.8,1)
  LDPhysics.SetPosition(block[i],500,GraphicsWindow.Height-20*i+10,0)
EndFor

bullet = Shapes.AddEllipse(8,8)
LDPhysics.AddMovingShape(bullet,0,1,1)
LDPhysics.SetPosition(bullet,50,200,0)
LDPhysics.SetVelocity(bullet,1000,0)

While ("True")
  LDPhysics.DoTimestep()
  Program.Delay(20)
EndWhile
```

Example 5 - Interact with the game loop

We use example 4, but reset the bullet when the left mouse button is clicked and reset the bricks when the right mouse button is clicked.

Capture the mouse click events

Set a flag when the left or right mouse click is registered.

```
leftMB = 0
rightMB = 0
GraphicsWindow.MouseDown = OnMouseDown

Sub OnMouseDown
  If (Mouse.IsLeftButtonDown) Then
    leftMB = 1
  ElseIf (Mouse.IsRightButtonDown) Then
    rightMB = 1
  EndIf
EndSub
```

Modified game loop

We need to either reset the bullet or blocks depending on the mouse click flag set. Note we have to reset the click flags after we have processed the command. We also give the bullet a random vertical (Y) component to its velocity.

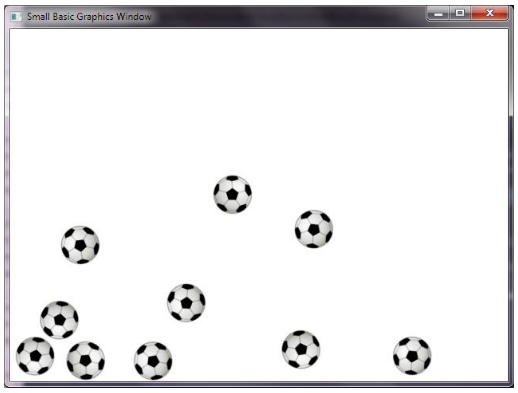
```
  If (leftMB = 1) Then
    LDPhysics.SetPosition(bullet,50,200,0)
    LDPhysics.SetVelocity(bullet,1000,Math.GetRandomNumber(401)-201)
    leftMB = 0
  EndIf
  If (rightMB = 1) Then
    For i = 1 To 20
      LDPhysics.SetPosition(block[i],500,GraphicsWindow.Height-20*i+10,0)
      LDPhysics.SetVelocity(block[i],0,0)
      LDPhysics.SetRotation(block[i],0)
    EndFor
    rightMB = 0
  EndIf
```

The whole thing

```
For i = 1 To 20
  block[i] = Shapes.AddRectangle(20,20)
  LDPhysics.AddMovingShape(block[i],0.3,0.8,1)
  LDPhysics.SetPosition(block[i],500,GraphicsWindow.Height-20*i+10,0)
EndFor

bullet = Shapes.AddEllipse(8,8)
LDPhysics.AddMovingShape(bullet,0,1,1)
LDPhysics.SetPosition(bullet,50,200,0)
LDPhysics.SetVelocity(bullet,1000,0)

leftMB = 0
rightMB = 0
GraphicsWindow.MouseDown = OnMouseDown

While ("True")
  If (leftMB = 1) Then
```

```
    LDPhysics.SetPosition(bullet,50,200,0)
    LDPhysics.SetVelocity(bullet,1000,Math.GetRandomNumber(101)-51)
    leftMB = 0
  EndIf
  If (rightMB = 1) Then
    For i = 1 To 20
      LDPhysics.SetPosition(block[i],500,GraphicsWindow.Height-20*i+10,0)
      LDPhysics.SetVelocity(block[i],0,0)
      LDPhysics.SetRotation(block[i],0)
    EndFor
    rightMB = 0
  EndIf
  LDPhysics.DoTimestep()
  Program.Delay(20)
EndWhile

Sub OnMouseDown
  If (Mouse.IsLeftButtonDown) Then
    leftMB = 1
  ElseIf (Mouse.IsRightButtonDown) Then
    rightMB = 1
  EndIf
EndSub
```

Example 6 – Using images



We use a downloaded football image and create 10 bouncing balls.

Create the objects in Small Basic and attach to the physics engine

Load an image.

```
image = ImageList.LoadImage(Program.Directory+"/football.png")
```

Tell the physics engine that the image shapes are to be circles.

```
LDPhysics.LoadImagesAsCircles = "True"
```

Create the Small Basic image shapes, add them to the physics engine, set their position randomly, and also set a random initial velocity.

```
For i = 1 To 10
  ball = Shapes.AddImage(image)
  LDPhysics.AddMovingShape(ball,0.2,0.9,1)
  LDPhysics.SetPosition(ball,Math.GetRandomNumber(500),Math.GetRandomNumber(300),0)
  LDPhysics.SetVelocity(ball,Math.GetRandomNumber(51)-101,0)
EndFor
```

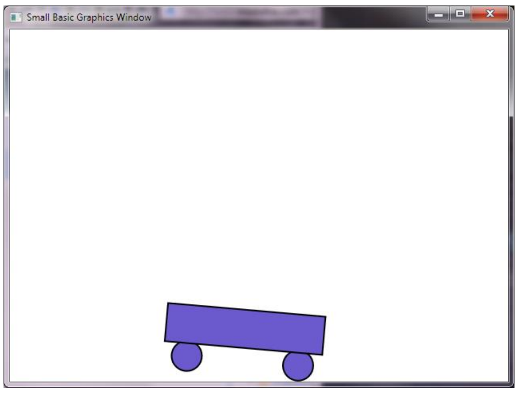
Create a game loop

This is just the same as before.

The whole thing

```
image = ImageList.LoadImage(Program.Directory+"/football.png")

LDPhysics.LoadImagesAsCircles = "True"

For i = 1 To 10
  ball = Shapes.AddImage(image)
  LDPhysics.AddMovingShape(ball,0.2,0.9,1)
  LDPhysics.SetPosition(ball,Math.GetRandomNumber(500),Math.GetRandomNumber(300),0)
  LDPhysics.SetVelocity(ball,Math.GetRandomNumber(51)-101,0)
EndFor

While ("True")
  LDPhysics.DoTimestep()
  Program.Delay(20)
EndWhile
```

Example 7 – A simple car



We use rotating attachments to allow the wheels to roll.

Create the objects in Small Basic

Create two wheels and the car body.

```
wheel1 = Shapes.AddEllipse(40,40)
wheel2 = Shapes.AddEllipse(40,40)
car = Shapes.AddRectangle(200,50)
```

Attach to the physics engine

Attach the wheels to the physics engine as moving shapes and position them.

```
LDPhysics.AddMovingShape(wheel1,1,0,1)
LDPhysics.AddMovingShape(wheel2,1,0,1)
LDPhysics.SetPosition(wheel1,180,400,0)
LDPhysics.SetPosition(wheel2,320,400,0)
```

Attach the car body to the physics engine as a moving shape and position it.

```
LDPhysics.AddMovingShape(car,0.3,0.5,1)
LDPhysics.SetPosition(car,250,360,0)
```

Attach the wheels to the car as rotating attachments.

```
LDPhysics.AttachShapesWithRotation(wheel1,car)
LDPhysics.AttachShapesWithRotation(wheel2,car)
```

Start the car moving to the right.

```
LDPhysics.SetVelocity(car,100,0)
```

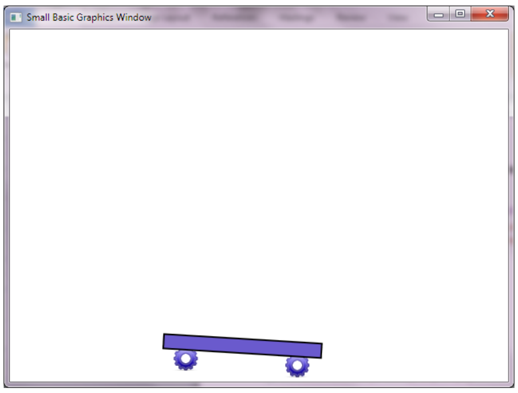
Create a game loop

This is just the same as before.

The whole thing

```
wheel1 = Shapes.AddEllipse(40,40)
wheel2 = Shapes.AddEllipse(40,40)
car = Shapes.AddRectangle(200,50)

LDPhysics.AddMovingShape(wheel1,1,0,1)
LDPhysics.AddMovingShape(wheel2,1,0,1)
LDPhysics.SetPosition(wheel1,180,400,0)
LDPhysics.SetPosition(wheel2,320,400,0)

LDPhysics.AddMovingShape(car,0.3,0.5,1)
LDPhysics.SetPosition(car,250,360,0)

LDPhysics.AttachShapesWithRotation(wheel1,car)
LDPhysics.AttachShapesWithRotation(wheel2,car)

LDPhysics.SetVelocity(car,100,0)

While ("True")
  LDPhysics.DoTimestep()
  Program.Delay(20)
EndWhile
```

Example 8 - A car that moves with arrow keys



Following example 7, we create the car, use images for wheels that are set to be circles in the physics engine.

Create the objects in Small Basic and attach to the physics engine

The wheels have a high friction value of 10 to reduce wheel spin.

```
image = ImageList.LoadImage(Program.Directory+"/gear_wheel.png")

wheel1 = Shapes.AddImage(image)
wheel2 = Shapes.AddImage(image)
car = Shapes.AddRectangle(200,20)

LDPhysics.LoadImagesAsCircles = "True"

LDPhysics.AddMovingShape(wheel1,10,0,1)
LDPhysics.AddMovingShape(wheel2,10,0,1)
LDPhysics.SetPosition(wheel1,180,400,0)
LDPhysics.SetPosition(wheel2,320,400,0)

LDPhysics.AddMovingShape(car,0.3,0.5,1)
LDPhysics.SetPosition(car,250,380,0)

LDPhysics.AttachShapesWithRotation(wheel1,car)
LDPhysics.AttachShapesWithRotation(wheel2,car)

LDPhysics.SetVelocity(car,100,0)
```

Create a game loop and keyboard controls

If the left key is pressed, a flag (left) is set and the rotation of the wheels is increased anti-clockwise by applying a negative torque and the car is given a small impulse left to get it going and up a bit (helps climb obstacles – add your own ramps, bricks or other obstacles).

Similarly for the right key.

```
left = 0
right = 0
GraphicsWindow.KeyDown = OnKeyDown
GraphicsWindow.KeyUp = OnKeyUp
```

Get the mass of the car and inertia of the wheels.

```
mass = LDPhysics.GetMass(car)
```

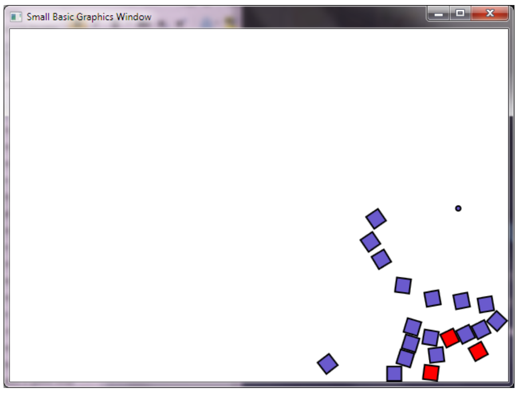
```
inerta = LDPhysics.GetInertia(wheel1)

While ("True")
  If (left = 1) Then
    LDPhysics.SetTorque(wheel1,-50*inerta)
    LDPhysics.SetTorque(wheel2,-50*inerta)
    LDPhysics.SetImpulse(car,-mass,-0.1*mass)
  EndIf
  If (right = 1) Then
    LDPhysics.SetTorque(wheel1,50*inerta)
    LDPhysics.SetTorque(wheel2,50*inerta)
    LDPhysics.SetImpulse(car,mass,-0.1*mass)
  EndIf
  LDPhysics.DoTimestep()
  Program.Delay(20)
EndWhile

Sub OnKeyDown
  k = GraphicsWindow.LastKey
  If (k = "Left") Then
    left = 1
  EndIf
  If (k = "Right") Then
    right = 1
  EndIf
EndSub

Sub OnKeyUp
  k = GraphicsWindow.LastKey
  If (k = "Left") Then
    left = 0
  EndIf
  If (k = "Right") Then
    right = 0
  EndIf
EndSub
```

The whole thing

```
image = ImageList.LoadImage(Program.Directory+"/gear_wheel.png")

wheel1 = Shapes.AddImage(image)
wheel2 = Shapes.AddImage(image)
car = Shapes.AddRectangle(200,20)

LDPhysics.LoadImagesAsCircles = "True"

LDPhysics.AddMovingShape(wheel1,10,0,1)
LDPhysics.AddMovingShape(wheel2,10,0,1)
LDPhysics.SetPosition(wheel1,180,400,0)
LDPhysics.SetPosition(wheel2,320,400,0)

LDPhysics.AddMovingShape(car,0.3,0.5,1)
LDPhysics.SetPosition(car,250,380,0)

LDPhysics.AttachShapesWithRotation(wheel1,car)
LDPhysics.AttachShapesWithRotation(wheel2,car)

LDPhysics.SetVelocity(car,100,0)
```

```
left = 0
right = 0
GraphicsWindow.KeyDown = OnKeyDown
GraphicsWindow.KeyUp = OnKeyUp

mass = LDPhysics.GetMass(car)
inerta = LDPhysics.GetInertia(wheel1)

While ("True")
  If (left = 1) Then
    LDPhysics.SetTorque(wheel1,-50*inerta)
    LDPhysics.SetTorque(wheel2,-50*inerta)
    LDPhysics.SetImpulse(car,-mass,-0.1*mass)
  EndIf
  If (right = 1) Then
    LDPhysics.SetTorque(wheel1,50*inerta)
    LDPhysics.SetTorque(wheel2,50*inerta)
    LDPhysics.SetImpulse(car,mass,-0.1*mass)
  EndIf
  LDPhysics.DoTimestep()
  Program.Delay(20)
EndWhile

Sub OnKeyDown
  k = GraphicsWindow.LastKey
  If (k = "Left") Then
    left = 1
  EndIf
  If (k = "Right") Then
    right = 1
  EndIf
EndSub

Sub OnKeyUp
  k = GraphicsWindow.LastKey
  If (k = "Left") Then
    left = 0
  EndIf
  If (k = "Right") Then
    right = 0
  EndIf
EndSub
```

Example 9 – Change the colour of hit blocks (Collision Detection)



We use example 4 as a starting point and modify it to change the colour of any block hit by the bullet.

Modify the game loop

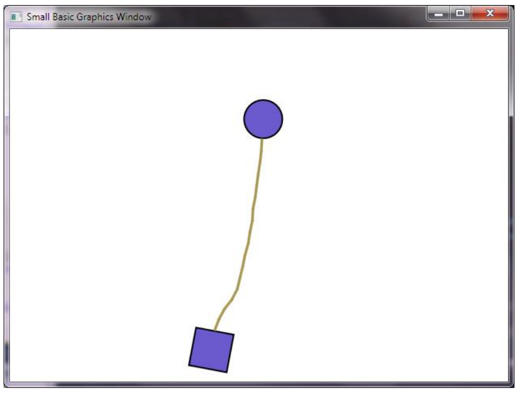
We use the **LDPhysics.GetCollisions** method to get a list of blocks hit by the bullet and change their colour using the **LDShapes.BrushColour** method.

```
While ("True")
  LDPhysics.DoTimestep()
  hits = LDPhysics.GetCollisions(bullet)
  For i = 1 To Array.GetItemCount(hits)
    If (hits[i] <> "Wall") Then
      LDShapes.BrushColour(hits[i],"Red")
    EndIf
  EndFor
  Program.Delay(20)
EndWhile
```

The whole thing

```
For i = 1 To 20
  block[i] = Shapes.AddRectangle(20,20)
  LDPhysics.AddMovingShape(block[i],0.3,0.8,1)
  LDPhysics.SetPosition(block[i],500,GraphicsWindow.Height-20*i+10,0)
EndFor

bullet = Shapes.AddEllipse(8,8)
LDPhysics.AddMovingShape(bullet,0,1,1)
LDPhysics.SetPosition(bullet,50,200,0)
LDPhysics.SetVelocity(bullet,1000,0)

While ("True")
  LDPhysics.DoTimestep()
  hits = LDPhysics.GetCollisions(bullet)
  For i = 1 To Array.GetItemCount(hits)
    If (hits[i] <> "Wall") Then
      LDShapes.BrushColour(hits[i],"Red")
    EndIf
  EndFor
  Program.Delay(20)
EndWhile
```

Example 10 - A balloon with a suspended rope and box (Mass and Forces)



Create the objects in Small Basic

These are just the balloon and box.

```
balloon = Shapes.AddEllipse(50,50)
box = Shapes.AddRectangle(50,50)
```

Attach to the physics engine

Add the balloon and box.

```
LDPhysics.AddMovingShape(balloon,0.3,0.5,1)
LDPhysics.SetPosition(balloon,100,100,0)
LDPhysics.AddMovingShape(box,0.3,0.5,1)
LDPhysics.SetPosition(box,300,300,0)
```

Create anchors on the balloon and box for the rope.

```
anchorBalloon = LDPhysics.AddMovingAnchor(100,125)
LDPhysics.AttachShapesWithRotation(balloon,anchorBalloon)
anchorBox = LDPhysics.AddMovingAnchor(300,275)
LDPhysics.AttachShapesWithRotation(box,anchorBox)
```

Now connect the rope to the anchors.

```
rope = LDPhysics.AddRope(anchorBalloon,anchorBox)
```

Finally, calculate the mass of the entire system.

```
boxMass = LDPhysics.GetMass(box)
balloonMass = LDPhysics.GetMass(balloon)
anchorBalloonMass = LDPhysics.GetMass(anchorBalloon)
anchorBoxMass = LDPhysics.GetMass(anchorBox)
ropeMass = LDPhysics.GetMass(rope)

totalMass = balloonMass+boxMass+ropeMass+anchorBalloonMass+anchorBoxMass
```

Create a game loop

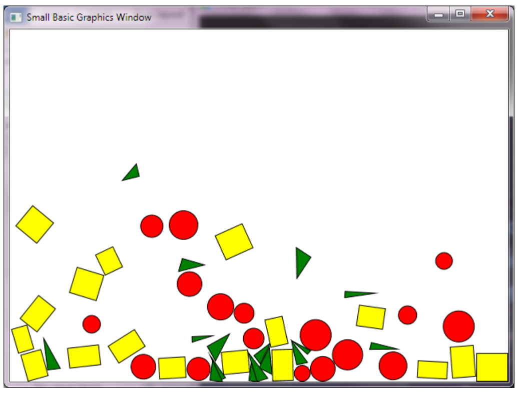
We set an upwards force equivalent to the total mass of the system acting against gravity (100 pixel/s^2), and apply it to the balloon each time-step.

```
While ("True")
  LDPhysics.SetForce(balloon,0,-100*totalMass)
  LDPhysics.DoTimestep()
  Program.Delay(20)
EndWhile
```

The whole thing

```
balloon = Shapes.AddEllipse(50,50)
box = Shapes.AddRectangle(50,50)

LDPhysics.AddMovingShape(balloon,0.3,0.5,1)
LDPhysics.SetPosition(balloon,100,100,0)
LDPhysics.AddMovingShape(box,0.3,0.5,1)
LDPhysics.SetPosition(box,300,300,0)

anchorBalloon = LDPhysics.AddMovingAnchor(100,125)
LDPhysics.AttachShapesWithRotation(balloon,anchorBalloon)
anchorBox = LDPhysics.AddMovingAnchor(300,275)
LDPhysics.AttachShapesWithRotation(box,anchorBox)

rope = LDPhysics.AddRope(anchorBalloon,anchorBox)

boxMass = LDPhysics.GetMass(box)
balloonMass = LDPhysics.GetMass(balloon)
anchorBalloonMass = LDPhysics.GetMass(anchorBalloon)
anchorBoxMass = LDPhysics.GetMass(anchorBox)
ropeMass = LDPhysics.GetMass(rope)

totalMass = balloonMass+boxMass+ropeMass+anchorBalloonMass+anchorBoxMass

While ("True")
  LDPhysics.SetForce(balloon,0,-100*totalMass)
  LDPhysics.DoTimestep()
  Program.Delay(20)
EndWhile
```

Example 11 - Mixed shapes



Demo showing rectangle, circle and triangle shapes; triangles were a struggle to integrate with Small Basic due to the way they move and rotate – try moving or rotating a triangle in Small Basic and figure out what it is doing.

The whole thing

```
GraphicsWindow.PenWidth = 1

For i = 1 To 50
  mode = Math.GetRandomNumber(3)
  If (mode = 1) Then
    GraphicsWindow.BrushColor = "Yellow"
    temp = Shapes.AddRectangle(20+Math.GetRandomNumber(20),20+Math.GetRandomNumber(20))
    LDPhysics.AddMovingShape(temp,0.5,0.8,1)
    LDPhysics.SetPosition(temp,Math.GetRandomNumber(600),50,0)
  ElseIf (mode = 2) Then
    GraphicsWindow.BrushColor = "Red"
    rad = 20+Math.GetRandomNumber(20)
    temp = Shapes.AddEllipse(rad,rad)
    LDPhysics.AddMovingShape(temp,0.5,0.8,1)
    LDPhysics.SetPosition(temp,Math.GetRandomNumber(600),50,0)
  Else
    GraphicsWindow.BrushColor = "Green"
    temp = Shapes.AddTriangle(0,0,20+Math.GetRandomNumber(20),0,0,2+Math.GetRandomNumber(20))
    LDPhysics.AddMovingShape(temp,0.5,0.8,1)
    LDPhysics.SetPosition(temp,Math.GetRandomNumber(600),50,0)
  EndIf
EndFor

While ("True")
  LDPhysics.DoTimestep()
  Program.Delay(20)
EndWhile
```

Example 12 – Attaching and grouping shapes

Two ways to attach shapes are available, **AttachShapes** and **GroupShapes**. The attach method works best for objects with moving parts, for example car wheels, but the attached shapes can move a bit when they collide with other shapes or boundaries – they are springy. The group method maintains completely rigid connections.

As well as, and probably a consequence of being springy, the attached shapes appear to slowly consume linear and rotational momentum, while the grouped shapes appear to slowly gain momentum.

A compound body is any collection of attached or grouped shapes.

The only problems when mixing attached and grouped shapes in the same compound body seem to be when the shapes have very different masses, which can be fixed by using appropriate densities.

In the terminology of Box2D, attached shapes are connected by Revolute and Distance joints, while grouping moves the shapes into one body.

Therefore using the grouping method, the position, velocity and angle for all grouped shapes will be the same since they are now the same body. The position of the compound body (**SetPosition** and **GetPosition**) refers to that of the second shape entered using the **GroupShapes** command (the first shape is 'added into' the second, maintaining their relative positions). The centre of mass and hence the centre of rotation is recalculated for the grouped compound body and will in general be different from its position (that of the shape to which others are grouped). This also means that the mass, inertia, force, torque etc. for any shape in the group now applies to the compound body as a whole. The collisions reported using **GetCollisions** still refer to the original shapes.

The recommended approach is:

- Use the **AttachShapes** method for any compound body that will have internal movement such as rotation, connected rope/chain etc.
- Use the **GroupShapes** method for a compound body which has no moving parts, but is just a collection of shapes.

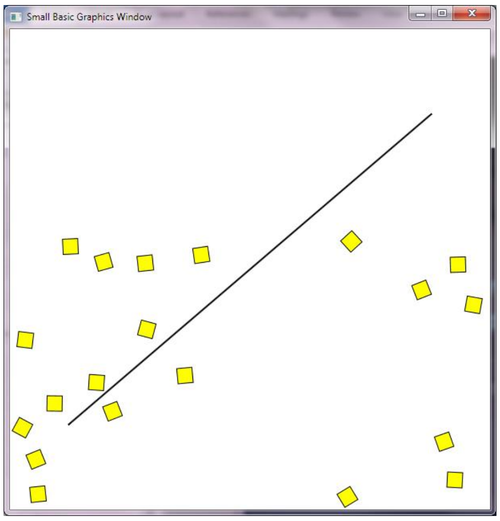
It is possible to attach and group shapes into complex bodies, but some care has to be taken to create a stable model. First group all shapes as required into compound bodies, then to attach (usually with rotation) these compound bodies.

The following example shows how a pencil can be made using both methods; in this case the grouping method is probably most appropriate.

The whole thing

```
GraphicsWindow.PenWidth = 0
GraphicsWindow.Width = 1000
GraphicsWindow.Height = 700
GraphicsWindow.Top = 0
GraphicsWindow.Left = 0

'Pencil 1
GraphicsWindow.BrushColor = "Yellow"
body1 = Shapes.AddRectangle(20,200)
GraphicsWindow.BrushColor = "Brown"
point1 = Shapes.AddTriangle(0,0,20,0,10,30)
GraphicsWindow.BrushColor = "Pink"
eraser1 = Shapes.AddRectangle(20,30)

LDPhysics.AddMovingShape(body1,0.3,0.3,1)
LDPhysics.SetPosition(body1,200,200,0)
LDPhysics.AddMovingShape(point1,0.3,0.5,1)
LDPhysics.SetPosition(point1,200,310,0)
LDPhysics.AddMovingShape(eraser1,0.8,0.8,1)
LDPhysics.SetPosition(eraser1,200,85,0)

LDPhysics.GroupShapes(point1,body1)
LDPhysics.GroupShapes(eraser1,body1)

'Pencil 2
GraphicsWindow.BrushColor = "Yellow"
body2 = Shapes.AddRectangle(20,200)
GraphicsWindow.BrushColor = "Brown"
point2 = Shapes.AddTriangle(0,0,20,0,10,30)
GraphicsWindow.BrushColor = "Pink"
eraser2 = Shapes.AddRectangle(20,30)

LDPhysics.AddMovingShape(body2,0.3,0.3,1)
LDPhysics.SetPosition(body2,800,200,0)
LDPhysics.AddMovingShape(point2,0.3,0.5,1)
LDPhysics.SetPosition(point2,800,310,0)
LDPhysics.AddMovingShape(eraser2,0.8,0.8,1)
LDPhysics.SetPosition(eraser2,800,85,0)

LDPhysics.AttachShapes(point2,body2)
LDPhysics.AttachShapes(eraser2,body2)

'A little sideways kick to break the symetry of the fall
LDPhysics.SetImpulse(body1,100,0)
LDPhysics.SetImpulse(body2,100,0)

While ("True")
  LDPhysics.DoTimestep()
  Program.Delay(10)
EndWhile
```

Example 13 – Defying gravity and user controlled interaction



If we don't interact with the physics engine, the shapes will fall, roll or swing etc. depending on what they are. If we want a paddle or other user movable object to interact with the moving shapes, then we cannot just move it since this will be seen like teleporting by the physics engine – the shape just disappeared from one place and reappeared in another. If this happens any interactions during the implied movement are lost.

The following is one approach to handling this and has a few steps.

- Create the user moving shape with a very large density (therefore large mass) and therefore won't move much when hit by other moving bodies.
- Apply a vertically up force to counteract gravity – the only remaining forces on the shape that can cause movement will be collisions from other shapes, but since they are light compared to our shape, these movements will be small.
- We can then reposition our shape to account for any small movements if we want it to be stationary – a teleport, but a very small one.
- We can then apply additional forces or torques to move or rotate our shape as required. It is important to apply a force or torque and not just reposition or rotate our shape manually if we want it to interact with the environment.

The whole thing

```
gw = 600
gh = 600
GraphicsWindow.Width = gw
GraphicsWindow.Height = gh

'Very heavy thin paddle, just smaller than the window
GraphicsWindow.PenWidth = 0
GraphicsWindow.BrushColor = "Black"
rod = Shapes.AddRectangle(gw-2,2)
LDPhysics.AddMovingShape(rod,0.3,0.5,1000000)
LDPhysics.SetPosition(rod,gw/2,gh/2,0)

'Some blocks
GraphicsWindow.PenWidth = 1
GraphicsWindow.BrushColor = "Yellow"
For i = 1 To 20
```

```
  block = Shapes.AddRectangle(20,20)
  LDPhysics.AddMovingShape(block,0.3,0.9,1)
  LDPhysics.SetPosition(block,gw/2,gh-i*20,0)
  LDPhysics.SetBullet(block)
EndFor

LDPhysics.TimeStep = 0.01 '100 fps

While ("True")
  start = Clock.ElapsedMilliseconds
  'Upwards force to counteract gravity
  LDPhysics.SetForce(rod,0,-LDPhysics.GetMass(rod)*100)
  'Reposition following any small movements using current angle
  LDPhysics.SetPosition(rod,gw/2,gh/2,LDPhysics.GetAngle(rod))
  'Apply rotation torque if we are rotating less than 45 deg/s
  If (LDPhysics.GetRotation(rod) < 45) Then '1/8 turn per sec
    LDPhysics.SetTorque(rod,1*LDPhysics.GetInertia(rod))
  EndIf
  LDPhysics.DoTimestep()
  'Delay upto timestep time
  delay = 1000*LDPhysics.TimeStep - (Clock.ElapsedMilliseconds - start)
  If (delay > 0) Then
    Program.Delay(delay)
  EndIf
EndWhile
```

An alternative method

The following is an alternative approach, attaching the paddle with rotation to a fixed central anchor.

```
gw = 600
gh = 600
GraphicsWindow.Width = gw
GraphicsWindow.Height = gh

'Very heavy thin paddle, just smaller than the window
GraphicsWindow.PenWidth = 0
GraphicsWindow.BrushColor = "Black"
rod = Shapes.AddRectangle(gw-2,2)
LDPhysics.AddMovingShape(rod,0.3,0.5,1000000)
LDPhysics.SetPosition(rod,gw/2,gh/2,0)

'Attach with rotation to a fixed anchor
anchor = LDPhysics.AddFixedAnchor(gw/2,gh/2)
LDPhysics.AttachShapesWithRotation(anchor,rod)

'Some blocks
GraphicsWindow.PenWidth = 1
GraphicsWindow.BrushColor = "Yellow"
For i = 1 To 20
  block = Shapes.AddRectangle(20,20)
  LDPhysics.AddMovingShape(block,0.3,0.9,1)
  LDPhysics.SetPosition(block,gw/2,gh-i*20,0)
  LDPhysics.SetBullet(block)
EndFor

LDPhysics.TimeStep = 0.01 '100 fps
```

```
While ("True")
  start = Clock.ElapsedMilliseconds
  'Apply rotation torque if we are rotating less than 45 deg/s
  If (LDPhysics.GetRotation(rod) < 45) Then '1/8 turn per sec
    LDPhysics.SetTorque(rod,1*LDPhysics.GetInertia(rod))
  EndIf
  LDPhysics.DoTimestep()
  'Delay upto timestep time
  delay = 1000*LDPhysics.TimeStep - (Clock.ElapsedMilliseconds - start)
  If (delay > 0) Then
    Program.Delay(delay)
  EndIf
EndWhile
```

Example 14 - Getting a shape from coordinates (Collision Detection)

We use example 6 and modify it so that when the user mouse clicks a ball it is given a vertical impulse.

Mouse click event handling

We need the standard Small Basic event raising procedure and set a flag (mouseDown) when the mouse is clicked.

```
mouseDown = 0
GraphicsWindow.MouseDown = OnMouseDown

Sub OnMouseDown
  mouseDown = 1
EndSub
```

We then need to act on this flag in the main game loop. We use the method **GetShapeAt**, which returns the shape (if any) at the input coordinates – in this case the mouse coordinates.

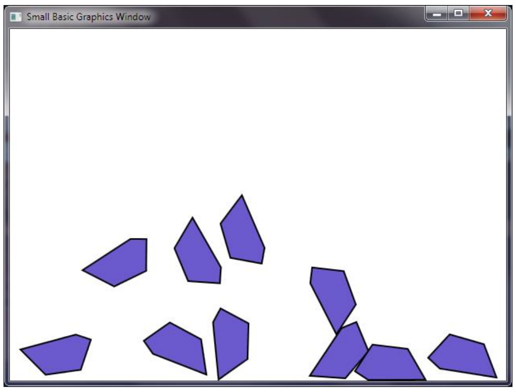
```
  If (mouseDown = 1) Then
    xM = GraphicsWindow.MouseX
    yM = GraphicsWindow.MouseY
    hit = LDPhysics.GetShapeAt(xM,yM)
    If (hit <> "") Then
      LDPhysics.SetImpulse(hit,0,-1000*LDPhysics.GetMass(hit))
    EndIf
    mouseDown = 0
  EndIf
```

The whole thing

```
image = ImageList.LoadImage(Program.Directory+"/football.png")

LDPhysics.LoadImagesAsCircles = "True"

For i = 1 To 10
  ball = Shapes.AddImage(image)
  LDPhysics.AddMovingShape(ball,0.2,0.9,1)
  LDPhysics.SetPosition(ball,Math.GetRandomNumber(500),Math.GetRandomNumber(300),0)
  LDPhysics.SetVelocity(ball,Math.GetRandomNumber(51)-101,0)
EndFor

mouseDown = 0
GraphicsWindow.MouseDown = OnMouseDown

While ("True")
  If (mouseDown = 1) Then
    xM = GraphicsWindow.MouseX
    yM = GraphicsWindow.MouseY
    hit = LDPhysics.GetShapeAt(xM,Ym)
    If (hit <> "") Then
      LDPhysics.SetImpulse(hit,0,-1000*LDPhysics.GetMass(hit))
    EndIf
    mouseDown = 0
  EndIf
  LDPhysics.DoTimestep()
  Program.Delay(20)
EndWhile
```

```
Sub OnMouseDown
  mouseDown = 1
EndSub
```

Example 15 - Using polygon shapes



Using the FC or LDShapes extension, it is possible to create and use general polygons, and as long as they are convex (all internal angles are less than 180 degrees) they can also be used by this extension. As an alternative it is possible to create compound shapes using the **GroupShapes** method, which is required in any case for shapes with concave boundaries or compound objects with varying properties such as density or restitution.

The whole thing

```
points[0]["X"] = 0
points[0]["Y"] = 0
points[1]["X"] = 40
points[1]["Y"] = 0
points[2]["X"] = 60
points[2]["Y"] = 40
points[3]["X"] = 40
points[3]["Y"] = 80
points[4]["X"] = 0
points[4]["Y"] = 20

For i = 1 To 10
  polygon = LDShapes.AddPolygon(points)
  LDPhysics.AddMovingShape(polygon,0.5,0.8,1)
  LDPhysics.SetPosition(polygon,60*i,100,0)
  LDPhysics.SetTorque(polygon,100*LDPhysics.GetInertia(polygon))
EndFor

While ("True")
  LDPhysics.DoTimestep()
  Program.Delay(20)
EndWhile
```

Example 16 - Sideways Scrolling

We extend Example 11 to use the Left and Right keys to pan left and right as the shapes fall.

No vertical boundaries

We can remove the vertical boundaries by setting them just larger than the GraphicsWindow.

```
LDPhysics.SetBoundaries(-1,1+GraphicsWindow.Width,0,GraphicsWindow.Height)
```

If we want to increase the off-screen region where the physics calculations are performed we could set the AABB at the start of the code (the first Physics command). The following sets the X (horizontal) region to ±1000 m or 10000 pixels using the default scaling of 10 pixel/m.

```
LDPhysics.SetAABB(-1000,1000,-100,200)
LDPhysics.Reset()
```

Key press events

We flag when the left and right keys are down using the KeyDown and KeyUp events.

```
panLeft = 0
panRight = 0
GraphicsWindow.keyDown = OnkeyDown
GraphicsWindow.keyUp = OnkeyUp
Sub OnkeyDown
  k = GraphicsWindow.LastKey
  If (k = "Left") Then
    panLeft = 1
  ElseIf (k = "Right") Then
    panRight = 1
  EndIf
EndSub
Sub OnkeyUp
  k = GraphicsWindow.LastKey
  If (k = "Left") Then
    panLeft = 0
  ElseIf (k = "Right") Then
    panRight = 0
  EndIf
EndSub
```

We write a subroutine to pan the display (10 pixels per step) when the key-press flag is set.

```
Sub handleEvents
  speed = 10
  If (panLeft = 1) Then
    LDPhysics.PanView(-speed,0)
  EndIf
  If (panRight = 1) Then
    LDPhysics.PanView(speed,0)
  EndIf
EndSub
```

The whole thing

```
LDPhysics.SetAABB(-1000,1000,-100,200)
LDPhysics.Reset()

GraphicsWindow.PenWidth = 1
```

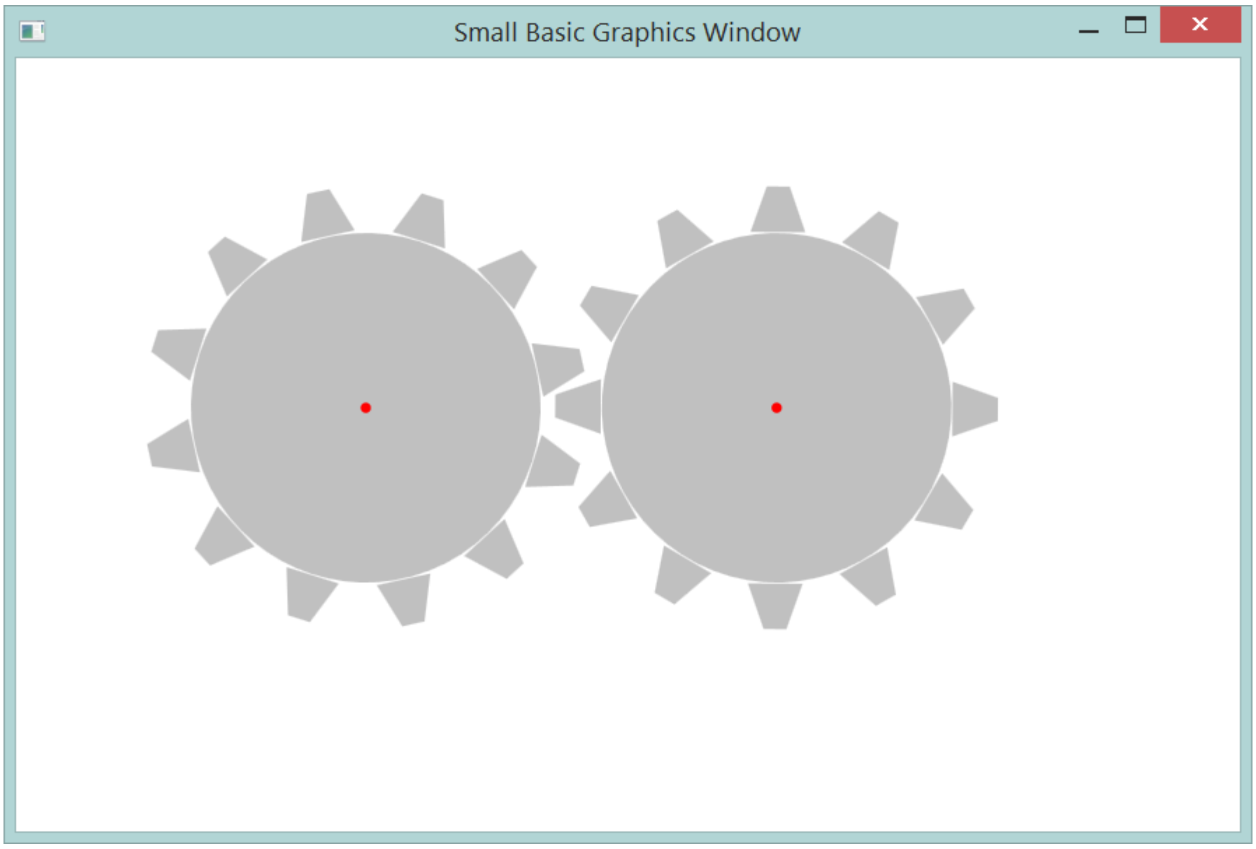
```
For i = 1 To 50
  mode = Math.GetRandomNumber(3)
  If (mode = 1) Then
    GraphicsWindow.BrushColor = "Yellow"
    temp = Shapes.AddRectangle(20+Math.GetRandomNumber(20),20+Math.GetRandomNumber(20))
    LDPhysics.AddMovingShape(temp,0.5,0.8,1)
    LDPhysics.SetPosition(temp,Math.GetRandomNumber(600),50,0)
  ElseIf (mode = 2) Then
    GraphicsWindow.BrushColor = "Red"
    rad = 20+Math.GetRandomNumber(20)
    temp = Shapes.AddEllipse(rad,rad)
    LDPhysics.AddMovingShape(temp,0.5,0.8,1)
    LDPhysics.SetPosition(temp,Math.GetRandomNumber(600),50,0)
  Else
    GraphicsWindow.BrushColor = "Green"
    temp = Shapes.AddTriangle(0,0,20+Math.GetRandomNumber(20),0,0,2+Math.GetRandomNumber(20))
    LDPhysics.AddMovingShape(temp,0.5,0.8,1)
    LDPhysics.SetPosition(temp,Math.GetRandomNumber(600),50,0)
  EndIf
EndFor

panLeft = 0
panRight = 0
GraphicsWindow.keyDown = OnkeyDown
GraphicsWindow.keyUp = OnkeyUp
Sub OnkeyDown
  k = GraphicsWindow.LastKey
  If (k = "Left") Then
    panLeft = 1
  ElseIf (k = "Right") Then
    panRight = 1
  EndIf
EndSub
Sub OnkeyUp
  k = GraphicsWindow.LastKey
  If (k = "Left") Then
    panLeft = 0
  ElseIf (k = "Right") Then
    panRight = 0
  EndIf
EndSub

LDPhysics.SetBoundaries(-1,1+GraphicsWindow.Width,0,GraphicsWindow.Height)

While ("True")
  handleEvents()
  LDPhysics.DoTimestep()
  Program.Delay(20)
EndWhile

Sub handleEvents
  speed = 10
  If (panLeft = 1) Then
    LDPhysics.PanView(-speed,0)
  EndIf
  If (panRight = 1) Then
    LDPhysics.PanView(speed,0)
  EndIf
EndSub
```

Other Samples

Here are some screenshots of other samples included.

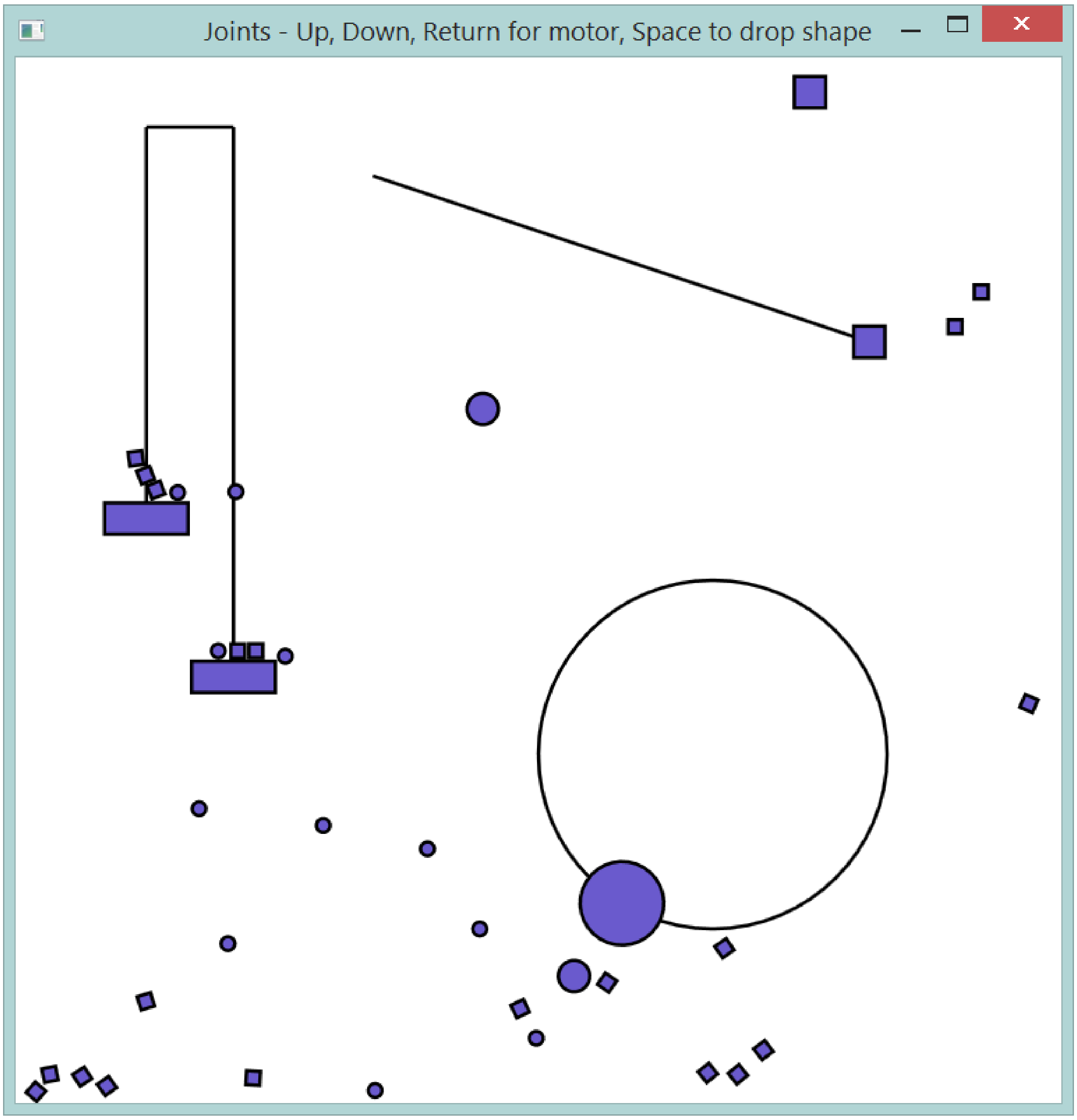
Cogs.sb

Intermeshing cogs that turn.



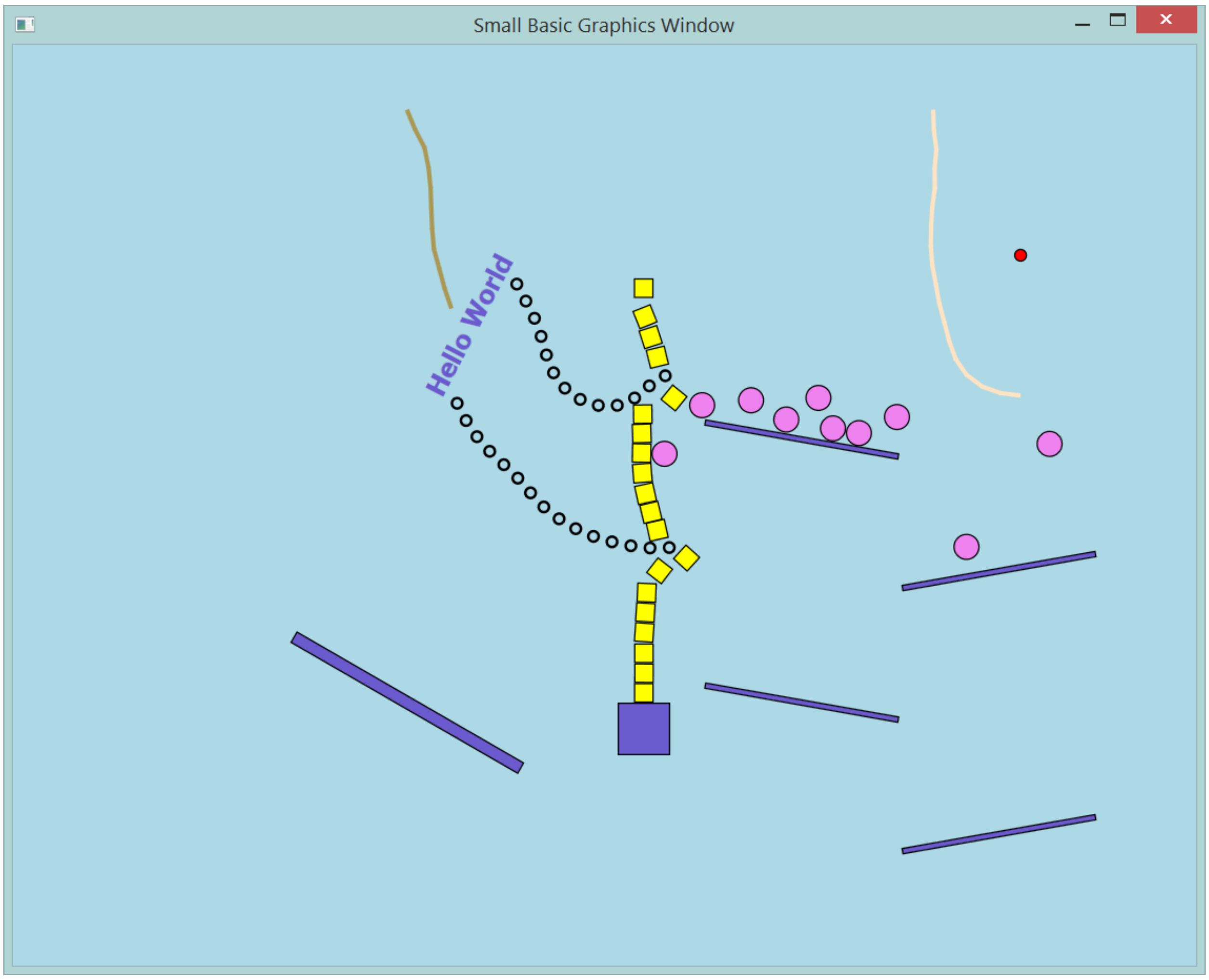
Joints.sb

Some advanced joints, including motors, pulleys and gears.



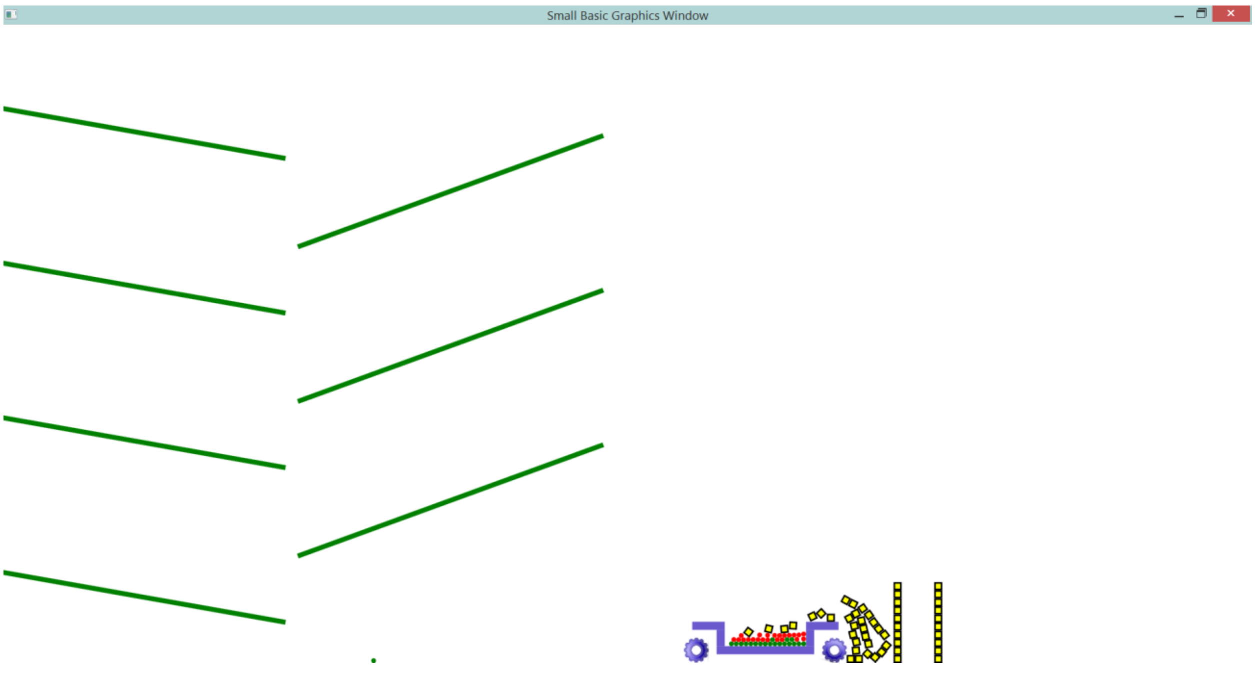
physics-sample.sb

Bouncing balls with ropes and chains.



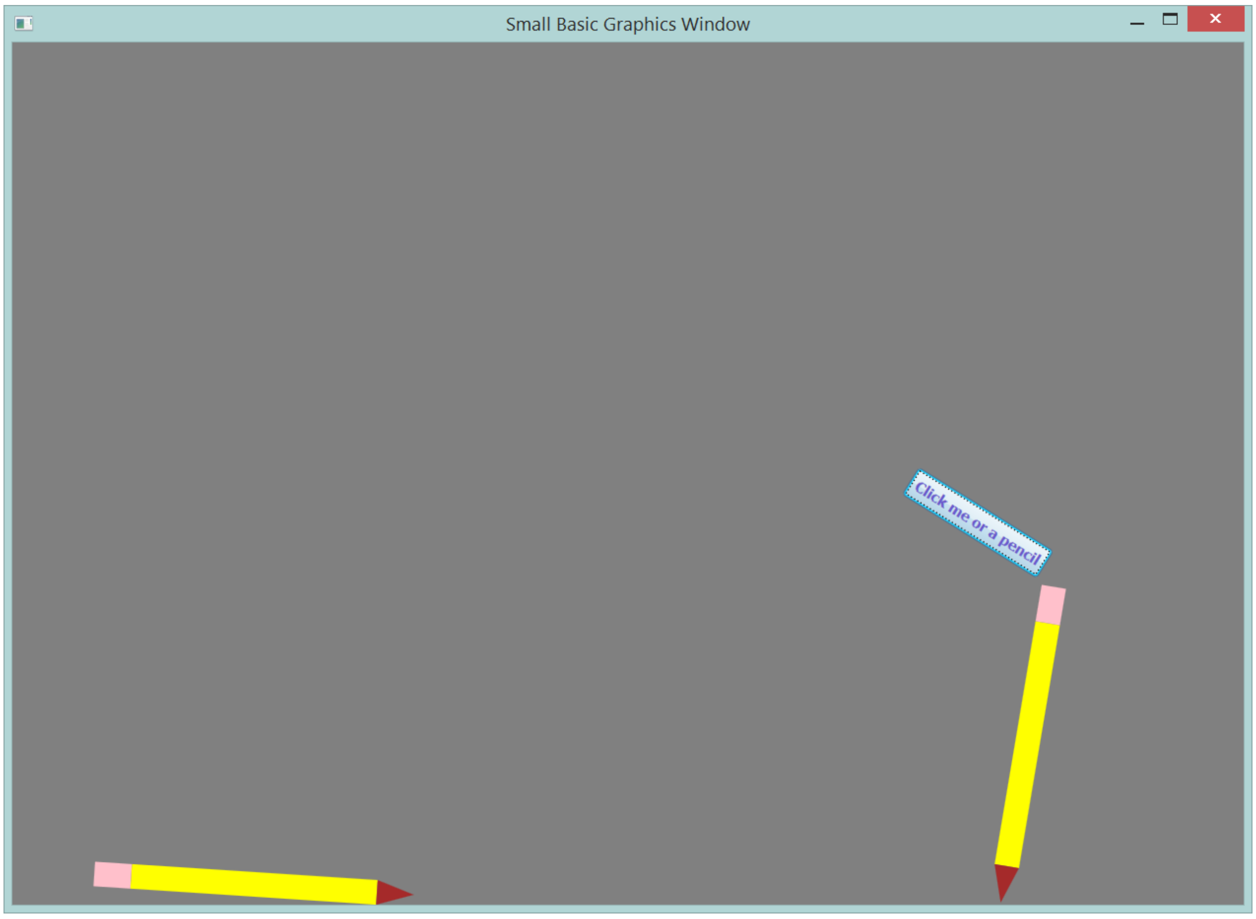
physics-sample-car.sb

A moving car.



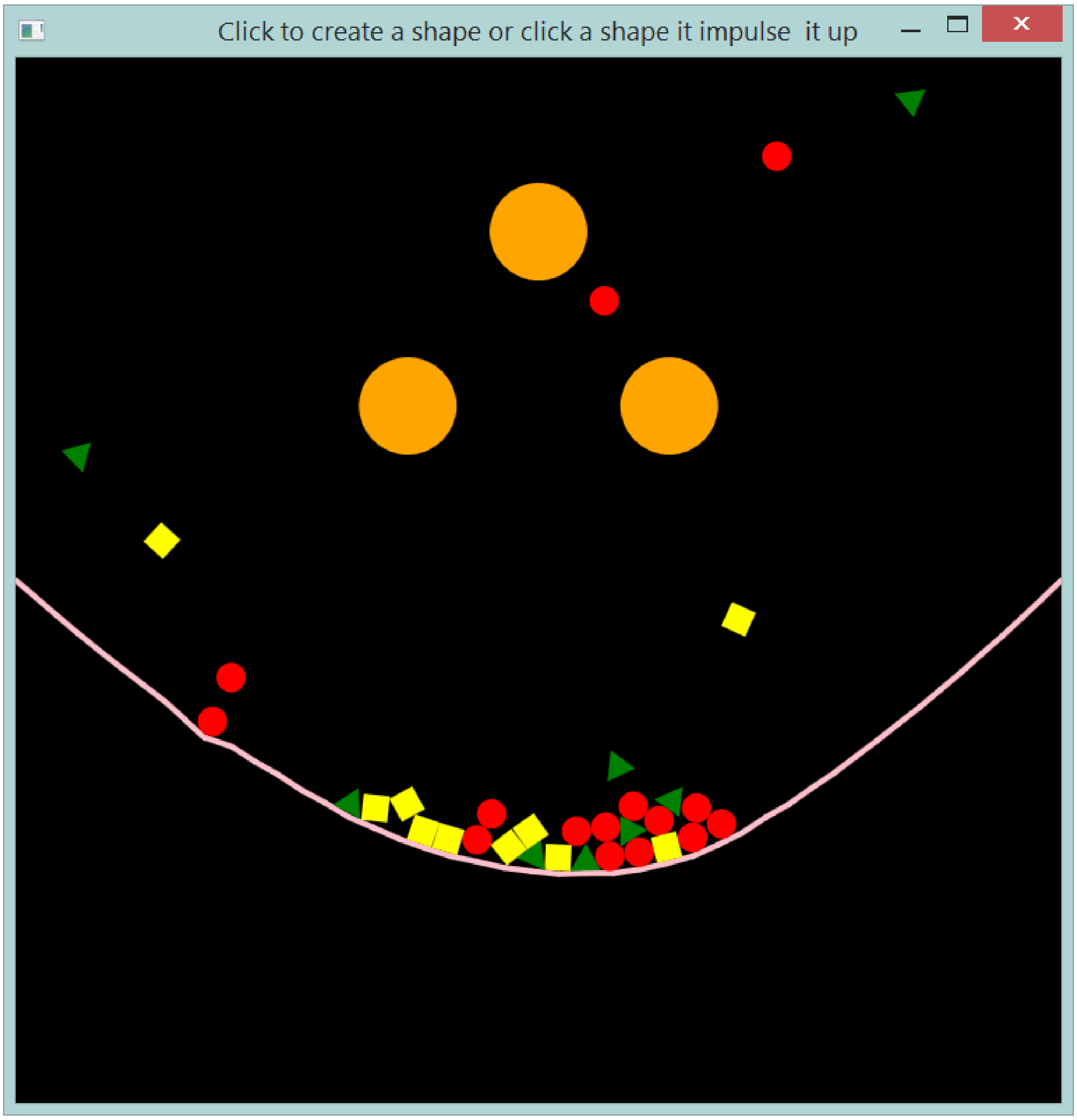
physics-sample-pencil.sb

Attached and grouped shapes.



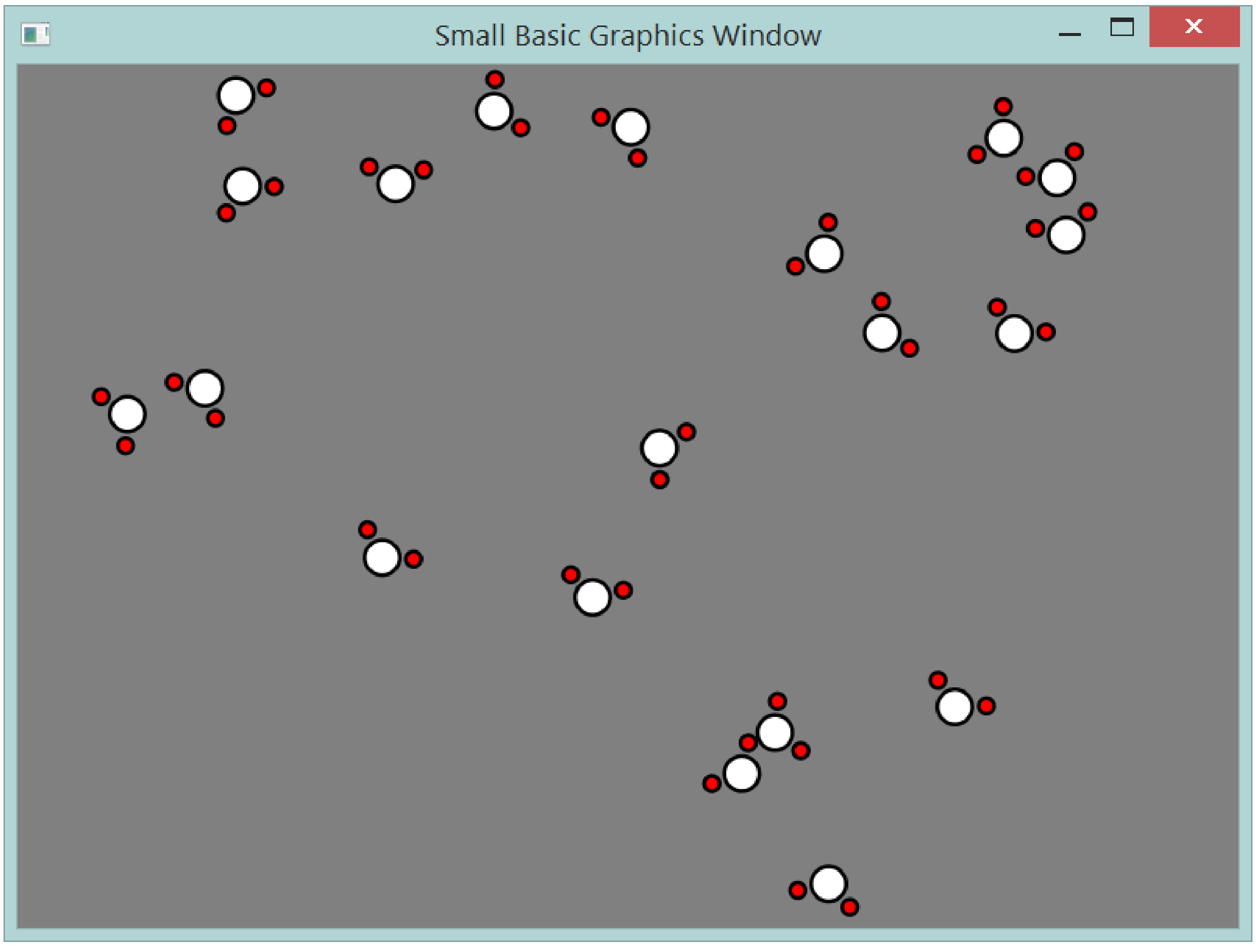
physics-sample-rope-bridge.sb

Many interacting shapes.



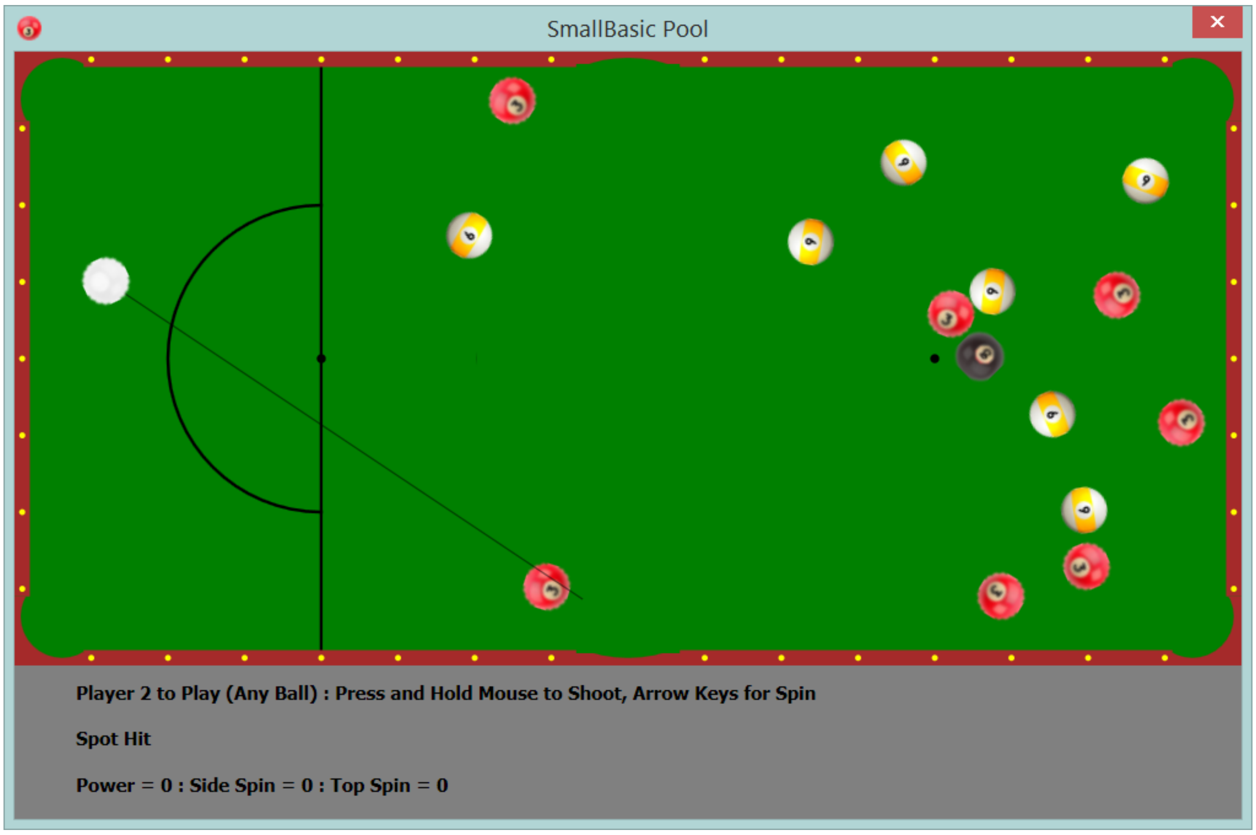
physics-sample-water.sb

Moving water molecules.



pool.sb

A full pool game.



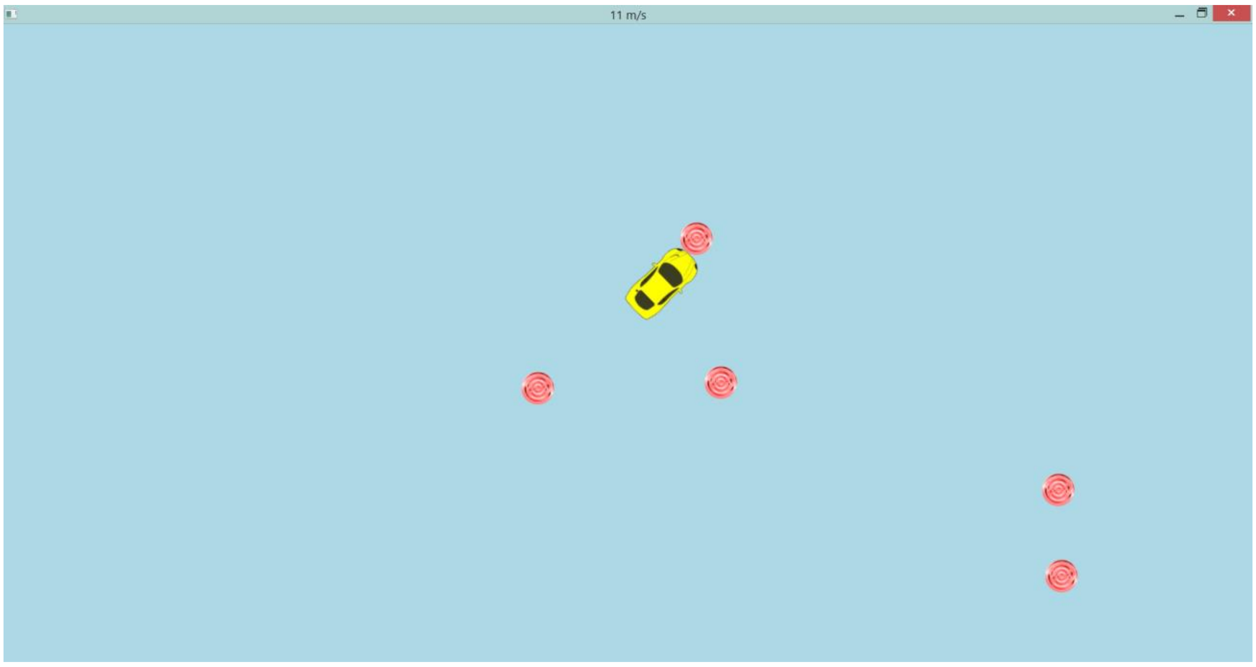
Player 2 to Play (Any Ball) : Press and Hold Mouse to Shoot, Arrow Keys for Spin

Spot Hit

Power = 0 : Side Spin = 0 : Top Spin = 0

TopDownCar.sb

A car with skid and crash detection in zero gravity, top-down.



Physics Extension API

The following is a list of the parameters and operations (Application Programming Interface) available with this extension.

ChainColour *PARAMETER*

The colour to be used for chains.

LoadImagesAsCircles *PARAMETER*

Toggle whether image shapes will be loaded as circles - "True" or "False" (default is "False").

MaxPolygonVertices *PARAMETER*

The physics engine maximum number of vertices on convex polygons (default 8).

MaxProxies *PARAMETER*

The physics engine maximum number of objects 'proxies' (default 1024).

PositionIterations *PARAMETER*

The physics engine position iterations (default 2).

RopeColour *PARAMETER*

The colour to be used for ropes.

Scaling *PARAMETER*

The physics engine scaling (pixel/m, default 10). It is not recommended to change this.

TimeStep *PARAMETER*

The physics engine time-step size (default 0.025).

VelocityIterations *PARAMETER*

The physics engine velocity iterations (default 6).

VelocityThreshold *PARAMETER*

The physics engine velocity threshold for inelastic collisions 'sticky walls' (default 1).

AddChain(shape1,shape2) *OPERATION*

Add a chain between two existing shapes.

shape1
First shape.

shape2
Second shape.

Returns
The chain name.

AddExplosion(posX,posY,power,damping,colour) *OPERATION*
Make an explosion, which consists of 50x20kg particles blast apart over 500ms.

posX
The X coordinate of the explosion.

posY
The Y coordinate of the explosion.

power
The explosion force, this is the initial velocity of the blast particles.

damping
A damping for the blast, the smaller this value the larger the blast range (default 10).

colour
An optional colour of the explosion particles ("" for none).

Returns
None.

AddFixedAnchor(posX,posY) *OPERATION*
Add a new small, transparent shape to be used as a fixed anchor point.

posX
The X coordinate of the anchor.

posY
The Y coordinate of the anchor.

Returns
The anchor shape name.

AddFixedShape(shapeName,friction,restitution) *OPERATION*
Add an existing Small Basic shape to the physics engine as a fixed (non-dynamic) shape with friction and restitution that affects shapes that hit it.

shapeName
The name of the shape.

friction
The shape friction (usually 0 to 1).

restitution
The shape restitution or bounciness (usually 0 to 1).

Returns
None.

AddInactiveShape(shapeName) *OPERATION*
Add an existing Small Basic shape to the physics engine as an inactive (non-dynamic and non-interacting) shape which only moves with the **PanView** method.

shapeName
The name of the shape.

Returns
None.

AddMovingAnchor(posX,posY) *OPERATION*
Add a new small, transparent and high density shape to be used as a moving anchor point.

posX
The X coordinate of the anchor.

posY
The Y coordinate of the anchor.

Returns
The anchor shape name.

AddMovingShape(shapeName,friction,restitution,density) *OPERATION*
Add an existing Small Basic shape to the physics engine as a moving (dynamic) shape.

shapeName
The name of the shape.

friction
The shape friction (usually 0 to 1).

restitution
The shape restitution or bounciness (usually 0 to 1). If a negative value is set for restitution, then the shape will be added with a very small size which may be used to add an inactive image that can be grouped within an irregular compound shape that matches the image boundary.

density
The shape density (default 1).

Returns
None.

AddRope(shape1,shape2) *OPERATION*
Add a rope between two existing shapes.

shape1
First shape.

shape2
Second shape.

Returns
The rope name.

AttachShapes(shape1,shape2) *OPERATION*
Connect two shapes to move together as one. The shapes are connected with a distance joint and may wobble a bit if they are hit.

shape1
The first shape name.

shape2
The second shape name.

Returns
None.

AttachShapesWithJoint(shape1,shape2,type,collide,parameters) *OPERATION*
Connect two shapes to move together as one with one of several joint types. These can be advanced and require reference to Box2D manual. In many cases it is best to prevent shape rotation for the joints to behave as desired. Multiple joints may also be applied to shapes. The methods use the initial shape positions, so set these first.

shape1
The first shape name.

shape2
The second shape name.

type
One of the following joint types.

- "Distance" - maintain a fixed distance between the shapes.
- "Gear" - link Prismatic or Revolute joints (previously created) of 2 shapes.
- "Line" - move the shapes along a line initially connecting the shapes.
- "Mouse" - move the shape to follow the mouse (both shape names should be the same).
- "Prismatic_H" - move shapes vertically along a line between the two shapes.
- "Prismatic_V" - move shapes horizontally along a line between the two shapes.

- "Pulley" - a pulley system, one shape moves up as the other moves down - position the shapes initially at the extreme points of the pulley motion.
- "Revolute" - the shapes can rotate about each other.

collide
The connected shapes can interact with each other "True" or "False" (default).

parameters
Optional parameters (default ""), multiple parameters are in an array.

- "Distance" - damping ratio (default 0)
- "Gear" - gear ratio, first joint, second joint (default 1, auto detect joints)
- "Line" - X direction, Y direction, lower translation, upper translation (default line connecting shapes, no limits)
- "Mouse" - max acceleration, damping ratio (default 10000, 0.7)
- "Prismatic_H" – X direction, Y direction, lower translation, upper translation (default 1,0, no limits)
- "Prismatic_V" - X direction, Y direction, lower translation, upper translation (default 0,1, no limits)
- "Pulley" - pulley ratio (block and tackle, default 1)
- "Revolute" - lower angle, upper angle (default no limits)

Returns
The joint name.

AttachShapesWithRotation(shape1,shape2) *OPERATION*
Connect two shapes to move together as one, but allow the shapes to rotate about each other.

shape1
The first shape name.

shape2
The second shape name.

Returns
None.

BoxShape(shapeName,x1,y1,x2,y2) *OPERATION*
Set a shape to remain within a box within the view. This is similar to PanView, except that the view pans automatically to keep the specified shape within a box region of the GraphicsWindow. Only one shape can be boxed. To unset shape box, set the shapeName to "".

shapeName
The shape to box or "".

x1

The left x coordinate of the box.

y1

The top x coordinate of the box.

x2

The right y coordinate of the box.

y2

The bottom y coordinate of the box.

Returns
None.

BrakeTire(shapeName) *OPERATION*
Apply a brake to a tire shape.

shapeName
The tire shape to brake.

Returns
None.

DetachJoint(jointName) *OPERATION*
Disconnect two shapes that were previously joined with a joint.

jointName
The joint name.

Returns
None.

DetachShapes(shape1,shape2) *OPERATION*
Disconnect two shapes that were previously attached.

shape1
The first shape name.

shape2
The second shape name.

Returns
None.

DisconnectShape(shapeName) *OPERATION*
Disconnect shape from the physics engine without deleting the shape.

shapeName
The shape name.

Returns
None.

DoTimestep() *OPERATION*
Perform a time-step update.

Returns
None.

FollowShapeX(shapeName) *OPERATION*
Set a shape to remain stationary at X position in the view. This is similar to PanView, except that the view pans automatically to keep the specified shape at a constant visual X location. Only one shape can be followed in X direction. To unset shape following, set the shapeName to "".

shapeName
The shape to follow or "".

Returns
None.

FollowShapeY(shapeName) *OPERATION*
Set a shape to remain stationary at Y position in the view. This is similar to PanView, except that the view pans automatically to keep the specified shape at a constant visual Y location. Only one shape can be followed in Y direction. To unset shape following, set the shapeName to "".

shapeName
The shape to follow or "".

Returns
None.

GetAllShapesAt(posX,posY) *OPERATION*
Get an array of all the physics engine shapes (if any) at the input coordinates. The coordinates for this method are the physics engine coordinates if panning is present.

posX
The X coordinate.

posY
The X coordinate.

Returns
An array of shape names or "".

GetAngle(shapeName) *OPERATION*
Get the angle of rotation for the shape.

shapeName
The shape name.

Returns
The angle of rotation in degrees.

GetCollisions(shapeName) *OPERATION*
Get an array of all the shapes that the specified shape collided with during the last DoTimestep().

shapeName
The shape to check for collisions.

Returns
An array of all the shapes collided with (may be empty "").

GetContacts(posX,posY,distance) *OPERATION*
Get a list of shapes that collided within a distance of a specified contact point.

posX
The X coordinate of a contact position to check.

posY
The Y coordinate of a contact position to check.

distance
A maximum distance from the contact point for the contact.

Returns
An array of contacts, with each contact being an array of 2 shape names.

GetInertia(shapeName) *OPERATION*
Get the moment of inertia of a shape.

shapeName
The shape name.

Returns
The inertia of the shape.

GetMass(shapeName) *OPERATION*
Get the mass of a shape.

shapeName
The shape name.

Returns
The mass of the shape.

GetPan() *OPERATION*
Get the current pan offset. See PanView, FollowShapeX(Y) and BoxShape. World coordinates = screen coordinates + pan offset.

Returns
A 2 element array with the current pan offset.

GetPosition(shapeName) *OPERATION*
Get the centre of the shape coordinates.

shapeName
The shape name.

Returns
A 2 element array with the shape centre position.

GetRotation(shapeName) *OPERATION*
Get the shape rotation speed.

shapeName
The shape name.

Returns
The angular rotation speed degrees/s.

GetShapeAt(posX,posY) *OPERATION*
Get the shape (if any) at the input coordinates. The coordinates for this method are the screen coordinates if panning is present.

posX
The X coordinate.

posY
The X coordinate.

Returns
The shape name at the input position or "".

GetTireInformation(shapeName) *OPERATION*
Get tire information, it includes:

- Skid (if this value exceeds the property AntiSkid, then the tire is skidding)

- Crash (the value is the speed of the impact)

shapeName
The tire shape.

Returns
An array of information, indexed by the information name, e.g. "Skid".

GetTireProperties(shapeName) *OPERATION*
Get tire properties, they include:

- AntiSkid (higher value reduces skid)
- Drag (higher value increases forward/backward drag)
- Brake (higher value increases braking power)
- Straighten (higher value restores steering more quickly)
- BrakeStraighten (higher value restores steering more quickly while braking)

shapeName
The tire shape.

Returns
An array of properties, indexed by the property name, e.g. "AntiSkid".

GetVelocity(shapeName) *OPERATION*
Get the velocity of the shape.

shapeName
The shape name.

Returns
A 2 element array with the shape velocity.

GroupShapes(shape1,shape2) *OPERATION*

Solidly group two shapes to move together as one. Shape1 is added to shape2 to act as one shape.

shape1
The first shape name.

shape2
The second shape name.

Returns
None.

Help() *OPERATION*
This function is just to display this help.

The extension uses Box2D (http://box2d.org) as an engine and provides an interface between it and the graphics capabilities of Small Basic.

Only shapes that are connected to the physics engine take part in the motion physics, for example you may add normal shapes (e.g. a gun and not connect it to the physics engine).

Once a shape is connected to the engine, it is best to only interact with it through the methods provided by the extension.

All positions are in the Small Basic **GraphicsWindow** pixels and refer to shape centres. Image and text shapes are treated as rectangles, and ellipses as circles; there is also triangle and convex polygon support, but not lines. Images may be treated as circles by setting the property **LoadImagesAsCircles** to "True".

One issue that Box2D has difficulty with is small fast moving objects that can 'tunnel' through other shapes without being deflected (see the **SetBullet** option).

Another problem is shapes of very different size and hence mass, especially large shapes when they are connected together. It may be necessary to modify the density for these (the Anchor options are an attempt to automate this a bit), otherwise the default density of 1 is good. Resist the temptation to connect too many shapes together.

It may be possible to improve the stability of some 'difficult' models using the **TimestepControl** settings, but the defaults look good for most cases.

Do not call the physics methods inside Small Basic event subroutines directly, rather set flags that can be processed in a main game loop.

There are sample Small Basic programs and a Getting Started Guide that comes with the extension dll - this is the best place to start.

Report bugs and problems to the Small Basic forum (http://social.msdn.microsoft.com/Forums/en-US/smallbasic/threads), but first simplify your Small Basic code to isolate the issue before providing a short 'runnable' code sample.

Returns
None.

MoveTire(shapeName,force) *OPERATION*
Move a tire shape, apply a forward or backward force.

shapeName
The tire shape to move.

force
The force to apply, positive is forward, negative is backward.

Returns
None.

PanView(panHorizontal,panVertical) *OPERATION*

Pan the camera view, including window boundaries.

panHorizontal
Pan in the horizontal direction (negative is left).

panVertical
Pan in the vertical direction (negative is up).

Returns
None.

RayCast(shapeName,angle,distance) *OPERATION*

Cast an invisible ray to detect the proximity of shapes.

shapeName
The shape to cast the ray from.

angle
The angle in degrees to check, this can also be an array of angles.

distance
A maximum distance to check.

Returns
An array of results, indexed by the shape name ("Wall" for a static obstacle) with a value equal to its distance. The shapes are sorted to list them nearest first.

If an array of input angles is used, then only the nearest shape for each angle is returned and the value is the angle, not the distance.

ReadJson(filename,scale,reverseY,stationary,offset,offsetY) *OPERATION*

Read in a json script compatible with R.U.B.E. and create a LDPhysics model. See https://www.iforce2d.net/rube for more details.

fileName
The full path to the json file to read.

scale
Scale all shapes, default 1 (no scaling).

reverseY
Reverse the Y direction up to down ("True" or "False").

staiionary
Set all shapes to be initially at rest, joint motors are still enabled ("True" or "False").

offestX
Add an x coordinate offset to all shapes.

offsetY
Add a y coordinate offset to all shapes, especially useful when reverseY is set.

Returns
A text array containing the LDPhysics commands used to create the model.

RemoveChain(shapeName) *OPERATION*
Remove a chain.

shapeName
The chain name.

Returns
None.

RemoveFrozen() *OPERATION*
Removes all frozen shapes - outside the AABB for the engine.

Returns
None.

RemoveRope(shapeName) *OPERATION*
Remove a rope.

shapeName
The rope name.

Returns
None.

RemoveShape(shapeName) *OPERATION*
Remove a shape.

shapeName
The name of the shape.

Returns
None.

Reset() *OPERATION*
Reset (delete all physics engine attached shapes).

Returns
None.

SetAABB(minX,maxX,minY,maxY) *OPERATION*
The physics engine AABB (axis-aligned bounding box). The units are the engine units of m. A Reset is required after setting. It is not recommended to change this.

minX
The left coordinate of the universe (default -100).

maxX
The right coordinate of the universe (default 200).

minY
The top coordinate of the universe (default -100).

maxY
The bottom coordinate of the universe (default 200).

Returns
None.

SetAngle(shapeName, angle) *OPERATION*
Reset the angle of rotation for a shape.

shapeName
The shape name.

angle
The angle of rotation in degrees.

Returns
None.

SetBoundaries(left,right,top,bottom) *OPERATION*
Set solid boundaries (positioning a boundary outside the GraphicsWindow removes it).

left
The left bounday X value.

right
The right bounday X value.

top
The top bounday Y value.

bottom
The bottom (ground) boundary Y value.

Returns
None.

SetBullet(shapeName) *OPERATION*
Set a shape as a bullet. This prevents 'tunnelling' of fast moving small objects at the expense of performance.

shapeName
The shape name.

Returns
None.

SetDamping(shapeName,linear,angular) *OPERATION*
Set a damping factor for a shape (default 0).

shapeName
The shape to modify.

linear
Linear damping factor.

angular
Angular damping factor.

Returns
None.

SetForce(shapeName,forceX,forceY) *OPERATION*
Set a force to apply to a shape (Remember F = ma).

shapeName
The shape to modify.

forceX
X component of the force.

forceY
Y component of the force.

Returns
None.

SetGravity(gravX,gravY) *OPERATION*

Set the gravity direction and magnitude (default 0,100).

gravX
The X component of gravity.

gravY
The Y component of gravity.

Returns
None.

SetGroup(shapeName,group,mask) *OPERATION*

Control which sprites interact (collide) with other shapes.

shapeName
The shape to modify.

group
The group that the current shape belongs to (default 0). This should be an integer between 0 and 7.

mask
An array of groups that this shape will collide with (default all groups).

Returns
None.

SetImpulse(shapeName,impulseX,impulseY) *OPERATION*

Set an impulse to a shape (a kick).

shapeName
The shape to modify.

impulseX
X component of the impulse.

impulseY
Y component of the impulse.

Returns
None.

SetJointMotor(jointName,speed,maxForce) *OPERATION*

Set a motor for selected joints (Line, Prismatic_H, Prismatic_V and Revolute).

jointName
The joint name.

speed
The desired motor speed.

maxForce
The maximum motor force (torque for Revolute). A zero value turns motor off.

Returns
None.

SetPosition(shapeName,posX,posY,angle) *OPERATION*

Reset shape position.

shapeName
The shape to modify.

posX
X component shape centre.

posY
Y component shape centre.

angle
The angle of rotation in degrees.

Returns
None.

SetRotation(shapeName,rotation) *OPERATION*

Set shape rotation speed.

shapeName
The shape to modify.

rotation
The angular rotation speed degrees/s.

Returns
None.

SetShapeGravity(shapeName,gravX,gravY) *OPERATION*

Set the gravity direction and magnitude for an individual shape (default 0,100).

shapeName
The shape to modify.

gravX
The X component of gravity.

gravY
The Y component of gravity.

Returns
None.

SetTire(shapeName) *OPERATION*

Set an object to act as a drivable tire for a top down game. Usually gravity will be 0 and the shape should already be added to the engine. The object should be initially positioned facing forward up on the display.

shapeName
The shape to make a tire.

Returns
None.

SetTireProperties(shapeName,properties) *OPERATION*
Set tire properties, they include:

- AntiSkid (higher value reduces skid)
- Drag (higher value increases forward/backward drag)
- Brake (higher value increases braking power)
- Straighten (higher value restores steering more quickly)
- BrakeStraighten (higher value restores steering more quickly while braking)

shapeName
The tire shape.

Returns
An array of one or more properties to set. The index is one of the properties (case sensitive) and the value is the property value.

SetTorque(shapeName,torque) *OPERATION*
Set a torque to a shape (a rotational kick).

shapeName
The shape to modify.

torque
The torque to apply.

Returns
None.

SetVelocity(shapeName,velX,velY) *OPERATION*
Set the velocity of a shape.

shapeName
The shape to modify.

velX
X component of the velocity.

velY
Y component of the velocity.

Returns
None.

TimestepControl(timestep,velocityIterations,positionIterations) *OPERATION*
Modify default time-step control parameters - also can be set using individual parameters.

timestep
Time-step (default 0.025).

velocityIterations
Velocity iterations (default 6).

positionIterations
Position iterations (default 2).

Returns
None.

TurnTire(shapeName,torque) *OPERATION*
Turn a tire shape, steer left or right.

shapeName
The tire shape to turn.

torque
The torque, rotation force to apply, positive is turn right, negative is turn left.

Returns
None.

ToggleMoving(shapeName) *OPERATION*
Toggle a moving shape to be fixed and vice-versa. This method also sets the rotation to be on or off to match if it is moving or fixed.

shapeName
The shape name.

Returns
None.

ToggleRotation(shapeName) *OPERATION*
Toggle a shape to not rotate and vice-versa. This method toggles the rotation property for fixed and moving shapes.

shapeName
The shape name.

Returns
None.

ToggleSensor(shapeName) *OPERATION*
Toggle a shape to act as a sensor and vice-versa. A sensor shape does not interact with other shapes, but still provides collision data.

shapeName
The shape name.

Returns
None.

UngroupShapes(shape1,shape2) *OPERATION*
Remove shape group pairing.

shape1
The first shape name.

shape2
The second shape name.

Returns
None.

UnsetBullet(shapeName) *OPERATION*
Unset a shape as a bullet. This reverts the shape to normal collision detection.

shapeName
The shape name.

Returns
None.

WakeAll() *OPERATION*
Wake all sleeping shapes - shapes sleep due to no applied forces or contacts. They wake automatically on any contact or applied force, so this action is rarely required.

Returns
None.

WriteJson(fileName) *OPERATION*
Write out a json script compatible with R.U.B.E. from current LDPhysics model. See https://www.iforce2d.net/rube for more details.

fileName
The full path to the json file to create.

Returns
None.