9

Physics in Games Development

In chapter, the following recipes will be covered:

* Using physics rules in your game
* Making things collide
* Installing and integrating Box2D
* Making a basic 2D physics game: Angry Nerds
* Making 3D Angry Nerds
* Creating a particle system
* Using ragdoll in your game

# Introduction

In modern games and games of the past, some amount of physics concept has always been added to increase the sense of realism. Although most of the physics in the games are an approximation or optimization of the actual physics rules, it does a good job in achieving the desired result. Physics in games are basically a rough implementation of the Newtonian laws of motion mixed with basic fundamentals of collision detection.

The trick however for a game developer is to write the code in such a way that it does not bottleneck the CPU and the game still runs at a desired framework. We will discuss some basic concepts that we would require to introduced physics in our game. For the sake of simplicity, we have integrated Box2D into our engine and along with a renderer (OpenGL), we will output some physics interaction with objects. For 3D physics, we will take the help of Bullet Physics SDK and display the desired result.

# Using Physics rule in your game

1. The first step to have physics in the game is to have the environment ready so that proper calculations can be applied to the bodies and the physics simulation can work on them

## Getting ready

To step through this recipe, you will need a machine running Windows and Visual Studio. No other prerequisites are required.

## How to do it...

In this recipe we will see how easy it is to add physics rule in our game.

* First set up all the objects in the game scene
* Make them have properties such that they have vector points and velocities.
* Assign bounding box or bounding circles, depending on the shape of the object
* Apply forces on each of the bodies
* Detect collision between them based on the shape

#include <Box2D/Collision/b2Collision.h>

#include <Box2D/Collision/Shapes/b2CircleShape.h>

#include <Box2D/Collision/Shapes/b2PolygonShape.h>

void b2CollideCircles(

b2Manifold\* manifold,

const b2CircleShape\* circleA, const b2Transform& xfA,

const b2CircleShape\* circleB, const b2Transform& xfB)

{

manifold->pointCount = 0;

b2Vec2 pA = b2Mul(xfA, circleA->m\_p);

b2Vec2 pB = b2Mul(xfB, circleB->m\_p);

b2Vec2 d = pB - pA;

float32 distSqr = b2Dot(d, d);

float32 rA = circleA->m\_radius, rB = circleB->m\_radius;

float32 radius = rA + rB;

if (distSqr > radius \* radius)

{

return;

}

manifold->type = b2Manifold::e\_circles;

manifold->localPoint = circleA->m\_p;

manifold->localNormal.SetZero();

manifold->pointCount = 1;

manifold->points[0].localPoint = circleB->m\_p;

manifold->points[0].id.key = 0;

}

void b2CollidePolygonAndCircle(

b2Manifold\* manifold,

const b2PolygonShape\* polygonA, const b2Transform& xfA,

const b2CircleShape\* circleB, const b2Transform& xfB)

{

manifold->pointCount = 0;

// Compute circle position in the frame of the polygon.

b2Vec2 c = b2Mul(xfB, circleB->m\_p);

b2Vec2 cLocal = b2MulT(xfA, c);

// Find the min separating edge.

int32 normalIndex = 0;

float32 separation = -b2\_maxFloat;

float32 radius = polygonA->m\_radius + circleB->m\_radius;

int32 vertexCount = polygonA->m\_count;

const b2Vec2\* vertices = polygonA->m\_vertices;

const b2Vec2\* normals = polygonA->m\_normals;

for (int32 i = 0; i < vertexCount; ++i)

{

float32 s = b2Dot(normals[i], cLocal - vertices[i]);

if (s > radius)

{

// Early out.

return;

}

if (s > separation)

{

separation = s;

normalIndex = i;

}

}

// Vertices that subtend the incident face.

int32 vertIndex1 = normalIndex;

int32 vertIndex2 = vertIndex1 + 1 < vertexCount ? vertIndex1 + 1 : 0;

b2Vec2 v1 = vertices[vertIndex1];

b2Vec2 v2 = vertices[vertIndex2];

// If the center is inside the polygon ...

if (separation < b2\_epsilon)

{

manifold->pointCount = 1;

manifold->type = b2Manifold::e\_faceA;

manifold->localNormal = normals[normalIndex];

manifold->localPoint = 0.5f \* (v1 + v2);

manifold->points[0].localPoint = circleB->m\_p;

manifold->points[0].id.key = 0;

return;

}

// Compute barycentric coordinates

float32 u1 = b2Dot(cLocal - v1, v2 - v1);

float32 u2 = b2Dot(cLocal - v2, v1 - v2);

if (u1 <= 0.0f)

{

if (b2DistanceSquared(cLocal, v1) > radius \* radius)

{

return;

}

manifold->pointCount = 1;

manifold->type = b2Manifold::e\_faceA;

manifold->localNormal = cLocal - v1;

manifold->localNormal.Normalize();

manifold->localPoint = v1;

manifold->points[0].localPoint = circleB->m\_p;

manifold->points[0].id.key = 0;

}

else if (u2 <= 0.0f)

{

if (b2DistanceSquared(cLocal, v2) > radius \* radius)

{

return;

}

manifold->pointCount = 1;

manifold->type = b2Manifold::e\_faceA;

manifold->localNormal = cLocal - v2;

manifold->localNormal.Normalize();

manifold->localPoint = v2;

manifold->points[0].localPoint = circleB->m\_p;

manifold->points[0].id.key = 0;

}

else

{

b2Vec2 faceCenter = 0.5f \* (v1 + v2);

float32 separation = b2Dot(cLocal - faceCenter, normals[vertIndex1]);

if (separation > radius)

{

return;

}

manifold->pointCount = 1;

manifold->type = b2Manifold::e\_faceA;

manifold->localNormal = normals[vertIndex1];

manifold->localPoint = faceCenter;

manifold->points[0].localPoint = circleB->m\_p;

manifold->points[0].id.key = 0;

}

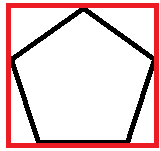
}

* Solve for the constraints
* Output the result

## How it works...

First step for a body to exhibit physics properties, is for it to be a rigid body. This is however not true if your body is supposed to have some kind of fluid physics as in the case of a plastic or any soft body. In that case we will have to setup the world differently as it is a far more complex problem. A rigid body in short is any object in world space, that will not deform even if there are external sources applied to it. Even in game engines like Unity or UE4, if you assign a body as rigid-body, it will automatically react based on the physics simulation property of the engine. After the rigid body is setup, we need to determine if the body is static or dynamic. This step is important as we can reduce the calculations immensely, if we know that the body is static. A dynamic body must be assigned velocities as well as vector positions.

After the above step is complete, the next step is to add colliders or bounding objects. These will actually be used for the calculations of collision points. For example, if we have a 3D model of a human, it is sometimes not very wise to use the exact body mesh for collision. Instead we could use a capsule, which is a cylinder with two half spheres on either end for the body and similar structure for the hands. In case of a 2D object, we make the choice between a circular bounding object or a box bounding object. The below diagram shows the object in black and the bounding box in red. We can now apply force or impulse to the objects.



IMG

The next step in the pipeline is to actually make the detection where two objects have collided. We will discuss more about this in the next recipe. But let’s say we have to detect whether a circle A has collided with another circle B, in most cases we just need the information whether they have collided rather than the exact point of contact. In that case we need to write some mathematical functions to detect that. Finally, we return the output and based on that, we write our logic for collision and finally display the result.

# Making things collide

A huge part of the physics system is to make things collide. We need to figure out whether the objects have collided and pass on the relevant information. In this recipe, we will figure what are the different techniques to do that.

## Getting ready

You need a Windows machine and a working copy of Visual Studio. No other pre-requisite is needed.

## How to do it...

In this recipe we will find out how easy it is to detect collisions.

#include <Box2D/Collision/b2Collision.h>

#include <Box2D/Collision/Shapes/b2PolygonShape.h>

// Find the max separation between poly1 and poly2 using edge normals from poly1.

static float32 b2FindMaxSeparation(int32\* edgeIndex,

const b2PolygonShape\* poly1, const b2Transform& xf1,

const b2PolygonShape\* poly2, const b2Transform& xf2)

{

int32 count1 = poly1->m\_count;

int32 count2 = poly2->m\_count;

const b2Vec2\* n1s = poly1->m\_normals;

const b2Vec2\* v1s = poly1->m\_vertices;

const b2Vec2\* v2s = poly2->m\_vertices;

b2Transform xf = b2MulT(xf2, xf1);

int32 bestIndex = 0;

float32 maxSeparation = -b2\_maxFloat;

for (int32 i = 0; i < count1; ++i)

{

// Get poly1 normal in frame2.

b2Vec2 n = b2Mul(xf.q, n1s[i]);

b2Vec2 v1 = b2Mul(xf, v1s[i]);

// Find deepest point for normal i.

float32 si = b2\_maxFloat;

for (int32 j = 0; j < count2; ++j)

{

float32 sij = b2Dot(n, v2s[j] - v1);

if (sij < si)

{

si = sij;

}

}

if (si > maxSeparation)

{

maxSeparation = si;

bestIndex = i;

}

}

\*edgeIndex = bestIndex;

return maxSeparation;

}

static void b2FindIncidentEdge(b2ClipVertex c[2],

const b2PolygonShape\* poly1, const b2Transform& xf1, int32 edge1,

const b2PolygonShape\* poly2, const b2Transform& xf2)

{

const b2Vec2\* normals1 = poly1->m\_normals;

int32 count2 = poly2->m\_count;

const b2Vec2\* vertices2 = poly2->m\_vertices;

const b2Vec2\* normals2 = poly2->m\_normals;

b2Assert(0 <= edge1 && edge1 < poly1->m\_count);

// Get the normal of the reference edge in poly2's frame.

b2Vec2 normal1 = b2MulT(xf2.q, b2Mul(xf1.q, normals1[edge1]));

// Find the incident edge on poly2.

int32 index = 0;

float32 minDot = b2\_maxFloat;

for (int32 i = 0; i < count2; ++i)

{

float32 dot = b2Dot(normal1, normals2[i]);

if (dot < minDot)

{

minDot = dot;

index = i;

}

}

// Build the clip vertices for the incident edge.

int32 i1 = index;

int32 i2 = i1 + 1 < count2 ? i1 + 1 : 0;

c[0].v = b2Mul(xf2, vertices2[i1]);

c[0].id.cf.indexA = (uint8)edge1;

c[0].id.cf.indexB = (uint8)i1;

c[0].id.cf.typeA = b2ContactFeature::e\_face;

c[0].id.cf.typeB = b2ContactFeature::e\_vertex;

c[1].v = b2Mul(xf2, vertices2[i2]);

c[1].id.cf.indexA = (uint8)edge1;

c[1].id.cf.indexB = (uint8)i2;

c[1].id.cf.typeA = b2ContactFeature::e\_face;

c[1].id.cf.typeB = b2ContactFeature::e\_vertex;

}

// Find edge normal of max separation on A - return if separating axis is found

// Find edge normal of max separation on B - return if separation axis is found

// Choose reference edge as min(minA, minB)

// Find incident edge

// Clip

// The normal points from 1 to 2

void b2CollidePolygons(b2Manifold\* manifold,

const b2PolygonShape\* polyA, const b2Transform& xfA,

const b2PolygonShape\* polyB, const b2Transform& xfB)

{

manifold->pointCount = 0;

float32 totalRadius = polyA->m\_radius + polyB->m\_radius;

int32 edgeA = 0;

float32 separationA = b2FindMaxSeparation(&edgeA, polyA, xfA, polyB, xfB);

if (separationA > totalRadius)

return;

int32 edgeB = 0;

float32 separationB = b2FindMaxSeparation(&edgeB, polyB, xfB, polyA, xfA);

if (separationB > totalRadius)

return;

const b2PolygonShape\* poly1; // reference polygon

const b2PolygonShape\* poly2; // incident polygon

b2Transform xf1, xf2;

int32 edge1; // reference edge

uint8 flip;

const float32 k\_tol = 0.1f \* b2\_linearSlop;

if (separationB > separationA + k\_tol)

{

poly1 = polyB;

poly2 = polyA;

xf1 = xfB;

xf2 = xfA;

edge1 = edgeB;

manifold->type = b2Manifold::e\_faceB;

flip = 1;

}

else

{

poly1 = polyA;

poly2 = polyB;

xf1 = xfA;

xf2 = xfB;

edge1 = edgeA;

manifold->type = b2Manifold::e\_faceA;

flip = 0;

}

b2ClipVertex incidentEdge[2];

b2FindIncidentEdge(incidentEdge, poly1, xf1, edge1, poly2, xf2);

int32 count1 = poly1->m\_count;

const b2Vec2\* vertices1 = poly1->m\_vertices;

int32 iv1 = edge1;

int32 iv2 = edge1 + 1 < count1 ? edge1 + 1 : 0;

b2Vec2 v11 = vertices1[iv1];

b2Vec2 v12 = vertices1[iv2];

b2Vec2 localTangent = v12 - v11;

localTangent.Normalize();

b2Vec2 localNormal = b2Cross(localTangent, 1.0f);

b2Vec2 planePoint = 0.5f \* (v11 + v12);

b2Vec2 tangent = b2Mul(xf1.q, localTangent);

b2Vec2 normal = b2Cross(tangent, 1.0f);

v11 = b2Mul(xf1, v11);

v12 = b2Mul(xf1, v12);

// Face offset.

float32 frontOffset = b2Dot(normal, v11);

// Side offsets, extended by polytope skin thickness.

float32 sideOffset1 = -b2Dot(tangent, v11) + totalRadius;

float32 sideOffset2 = b2Dot(tangent, v12) + totalRadius;

// Clip incident edge against extruded edge1 side edges.

b2ClipVertex clipPoints1[2];

b2ClipVertex clipPoints2[2];

int np;

// Clip to box side 1

np = b2ClipSegmentToLine(clipPoints1, incidentEdge, -tangent, sideOffset1, iv1);

if (np < 2)

return;

// Clip to negative box side 1

np = b2ClipSegmentToLine(clipPoints2, clipPoints1, tangent, sideOffset2, iv2);

if (np < 2)

{

return;

}

// Now clipPoints2 contains the clipped points.

manifold->localNormal = localNormal;

manifold->localPoint = planePoint;

int32 pointCount = 0;

for (int32 i = 0; i < b2\_maxManifoldPoints; ++i)

{

float32 separation = b2Dot(normal, clipPoints2[i].v) - frontOffset;

if (separation <= totalRadius)

{

b2ManifoldPoint\* cp = manifold->points + pointCount;

cp->localPoint = b2MulT(xf2, clipPoints2[i].v);

cp->id = clipPoints2[i].id;

if (flip)

{

// Swap features

b2ContactFeature cf = cp->id.cf;

cp->id.cf.indexA = cf.indexB;

cp->id.cf.indexB = cf.indexA;

cp->id.cf.typeA = cf.typeB;

cp->id.cf.typeB = cf.typeA;

}

++pointCount;

}

}

manifold->pointCount = pointCount;

}

## How it works...

Assuming the objects in the scene are already setup as rigid body, and the proper impulses are added to each, the next step is to detect the collisions.

In games, collision detection usually happens in two phases. The first phase is called the broad-phase collision and the next phase is called the narrow-phase collision. Broad phase is less expensive as it deals with a concept of, “What bodies are most likely to collide”. Narrow bodies are more expensive because they actually compare each body for collision. In a games environment it is not feasible to have everything in the narrow phase. Hence most of it is done in the broad phase. Broad phase algorithms work with sweep and prune (sort and prune) or Space partition tree. In sweep and prune technique, all the lower ends and upper ends of the bounding boxes of the solids are sorted and checked for intersection. After that it is sent to a more detailed check in the narrow phase. So in this method, we need to update the bounding box of the solid ever time it is oriented. The other technique used is BSP. We have already discussed BSP in previous chapters. We need to partition the scene in such a way that in each subdivision, only a certain number of objects can collide. In the narrow phase collision, a more pixel perfect collision detection algorithm is applied.

There are various ways to check for collision. It entirely depends on the shape that is acting act as the bounding box. Also it is important to understand how the bounding box is aligned. In a normal scenario a bounding box would be axes aligned and would be referred to as AABB. To detect if two Box2D bounding boxes have collided, we would have to do the following.

bool BoxesIntersect(const Box2D &a, const Box2D &b)

{

if (a.max.x < b.min.x) return false; // a is left of b

if (a.min.x > b.max.x) return false; // a is right of b

if (a.max.y < b.min.y) return false; // a is above b

if (a.min.y > b.max.y) return false; // a is below b

return true; // boxes overlap

}

We can then extend this to detect more complex shapes of rectangles, circles, line and other polygons. If we are writing our own 2D Physics engine, then we would have to write function for each combination of shape intersecting with one another. If we use a physics engine like Box2D or PhysX, these functions would be already written for us and we would have to just use them properly and consistently.

# Installing and Integrating Box2D

To be able to work with 2D physics, one great open source physics engine is Box2D. This comes inbuilt with lots of functions which would be common for any 2D game, so we do not have to reinvent the wheel and write them again.

## Getting ready

1. You need to have a working Windows machine.

## How to do it...

* Go to the following URL <http://box2d.org/>
* Browse to <http://box2d.org/downloads/>
* Download or clone the latest copy from the GitHub
* Build the solution in your version of Visual Studio. Some of the projects may not work as they were built in different versions of Visual Studio
* If that throws any error, clean the solution, delete the bin folder and rebuild it again
* After it rebuilds successfully, run the TestBed projects
* If you can run the application successfully, Box2D has been integrated.

## How it works...

Box2D is a physics engine built entirely in C++. As they have given us access to the source code, it means we can build it as well from scratch and check for ourselves how each function is written. As the project is hosted on GitHub, every time a new development is done, we can clone it to get updated with all the latest code.

In the solution, Box2D already has a project called TestBed which has loads of sample applications which can be run. It is actually a collection of loads of different types of applications. Test Entries is the entry point of all the applications. It is a long array of the different applications that we want rendered in the TestBed project. The array contains the name of the application and the static function to initialise the world.

Finally, the output of the physics simulation is fed to the renderer which in this case is OpenGL and it draws the scene for us.

# Making a basic 2D game

Every 2D game is different. However, we can generalize the physics functions that is going to be used in most 2D games. In this recipe, we will create a basic scene using Box2D’s inbuilt functions and the TestBed project. The scene will be mimicking one of the most popular 2D games of our times, Angry Birds.

## Getting ready

For this recipe, you will need a Windows machine and an installed version of Visual Studio. No other pre requisite is needed.

## How to do it...

1. In this recipe we will find out how easy it is to add a bare bone architecture for a 2D game using Box2D.
2. class Tiles : public Test
3. {
4. public:
5. enum
6. {
7. e\_count = 10
8. };
9. Tiles()
10. {
11. m\_fixtureCount = 0;
12. b2Timer timer;
13. {
14. float32 a = 1.0f;
15. b2BodyDef bd;
16. bd.position.y = -a;
17. b2Body\* ground = m\_world->CreateBody(&bd);
18. #if 1
19. int32 N = 200;
20. int32 M = 10;
21. b2Vec2 position;
22. position.y = 0.0f;
23. for (int32 j = 0; j < M; ++j)
24. {
25. position.x = -N \* a;
26. for (int32 i = 0; i < N; ++i)
27. {
28. b2PolygonShape shape;
29. shape.SetAsBox(a, a, position, 0.0f);
30. ground->CreateFixture(&shape, 0.0f);
31. ++m\_fixtureCount;
32. position.x += 2.0f \* a;
33. }
34. position.y -= 2.0f \* a;
35. }
36. #else
37. int32 N = 200;
38. int32 M = 10;
39. b2Vec2 position;
40. position.x = -N \* a;
41. for (int32 i = 0; i < N; ++i)
42. {
43. position.y = 0.0f;
44. for (int32 j = 0; j < M; ++j)
45. {
46. b2PolygonShape shape;
47. shape.SetAsBox(a, a, position, 0.0f);
48. ground->CreateFixture(&shape, 0.0f);
49. position.y -= 2.0f \* a;
50. }
51. position.x += 2.0f \* a;
52. }
53. #endif
54. }
55. {
56. float32 a = 1.0f;
57. b2PolygonShape shape;
58. shape.SetAsBox(a, a);
60. b2Vec2 x(-7.0f, 0.75f);
61. b2Vec2 y;
62. b2Vec2 deltaX(1.125f, 2.5f);
63. b2Vec2 deltaY(2.25f, 0.0f);
64. for (int32 i = 0; i < e\_count; ++i)
65. {
66. y = x;
67. for (int32 j = i; j < e\_count; ++j)
68. {
69. b2BodyDef bd;
70. bd.type = b2\_dynamicBody;
71. bd.position = y;
72. //if (i == 0 && j == 0)
73. //{
74. // bd.allowSleep = false;
75. //}
76. //else
77. //{
78. // bd.allowSleep = true;
79. //}
80. b2Body\* body = m\_world->CreateBody(&bd);
81. body->CreateFixture(&shape, 5.0f);
82. ++m\_fixtureCount;
83. y += deltaY;
84. }
85. x += deltaX;
86. }
87. }
88. m\_createTime = timer.GetMilliseconds();
89. }
90. void Step(Settings\* settings)
91. {
92. const b2ContactManager& cm = m\_world->GetContactManager();
93. int32 height = cm.m\_broadPhase.GetTreeHeight();
94. int32 leafCount = cm.m\_broadPhase.GetProxyCount();
95. int32 minimumNodeCount = 2 \* leafCount - 1;
96. float32 minimumHeight = ceilf(logf(float32(minimumNodeCount)) / logf(2.0f));
97. g\_debugDraw.DrawString(5, m\_textLine, "dynamic tree height = %d, min = %d", height, int32(minimumHeight));
98. m\_textLine += DRAW\_STRING\_NEW\_LINE;
99. Test::Step(settings);
100. g\_debugDraw.DrawString(5, m\_textLine, "create time = %6.2f ms, fixture count = %d",
101. m\_createTime, m\_fixtureCount);
102. m\_textLine += DRAW\_STRING\_NEW\_LINE;
103. //b2DynamicTree\* tree = &m\_world->m\_contactManager.m\_broadPhase.m\_tree;
104. //if (m\_stepCount == 400)
105. //{
106. // tree->RebuildBottomUp();
107. //}
108. }
109. static Test\* Create()
110. {
111. return new Tiles;
112. }
113. int32 m\_fixtureCount;
114. float32 m\_createTime;
115. };

#endif

## How it works...

In this example, we are using the Box2D engine to calculate the physics. The main class of Test Entries as described previously is used to store the name of the application and the static create method. In this case, the name of the application is called Tiles. In the tiles application, we have created a physics world using the Box2D shapes and functions. The pyramid of ties is created with the help of boxes. These boxes are dynamic in nature which means they will react and move based on forces applied to it. The base or the ground is also made of tiles. However, those tiles are stationary and do not move. We assign position and velocity for all the tiles that make up the ground and the pyramid. It is not practical to individually assign position and velocity too each tile. Hence we do this with an iteration loop.

After the scene is built, we can interact with the pyramid using the mouse click. From the GUI, also other properties can be switched on or off. Pressing Spacebar also triggers a ball at a random position and it will destroy the formation of the tiles much like angry Birds. We can also write a logic to make all the tiles disappear which are colliding with the ground and add points to the score every time that happens, and we have ourselves a small 2D Angry Birds clone.

# Making a 3D game

Not much changes when we shift our focus from physics in 2Dimension to physics in 3Dimension. We now need to worry about another dimension. Like mentioned in the previous recipes, we still need to maintain the environment so that it follows Newtonian rules and solves constraints. There are lot of things to which may go wrong while rotating the body in 3D space. In this recipe, we will see a very basic implementation of the 3D physics using the Bullet engine SDK.

## Getting ready

1. For this recipe, you will need a Windows machine and an installed version Visual Studio.

## How to do it...

In this recipe, we will see how easy it is to write a physics world in 3D.

**Broad Phase Collision**

void b3DynamicBvhBroadphase::getAabb(int objectId,b3Vector3& aabbMin, b3Vector3& aabbMax ) const

{

const b3DbvtProxy\* proxy=&m\_proxies[objectId];

aabbMin = proxy->m\_aabbMin;

aabbMax = proxy->m\_aabbMax;

}

1. **Narrow Phase Collision**
2. **void b3CpuNarrowPhase::computeContacts(b3AlignedObjectArray<b3Int4>& pairs, b3AlignedObjectArray<b3Aabb>& aabbsWorldSpace, b3AlignedObjectArray<b3RigidBodyData>& bodies)**
3. **{**
4. **int nPairs = pairs.size();**
5. **int numContacts = 0;**
6. **int maxContactCapacity = m\_data->m\_config.m\_maxContactCapacity;**
7. **m\_data->m\_contacts.resize(maxContactCapacity);**
8. **for (int i=0;i<nPairs;i++)**
9. **{**
10. **int bodyIndexA = pairs[i].x;**
11. **int bodyIndexB = pairs[i].y;**
12. **int collidableIndexA = bodies[bodyIndexA].m\_collidableIdx;**
13. **int collidableIndexB = bodies[bodyIndexB].m\_collidableIdx;**
14. **if (m\_data->m\_collidablesCPU[collidableIndexA].m\_shapeType == SHAPE\_SPHERE &&**
15. **m\_data->m\_collidablesCPU[collidableIndexB].m\_shapeType == SHAPE\_CONVEX\_HULL)**
16. **{**
17. **// computeContactSphereConvex(i,bodyIndexA,bodyIndexB,collidableIndexA,collidableIndexB,&bodies[0],**
18. **// &m\_data->m\_collidablesCPU[0],&hostConvexData[0],&hostVertices[0],&hostIndices[0],&hostFaces[0],&hostContacts[0],nContacts,maxContactCapacity);**
19. **}**
20. **if (m\_data->m\_collidablesCPU[collidableIndexA].m\_shapeType == SHAPE\_**
21. **m\_data->m\_contacts.resize(numContacts);**

**<. . . . . . . More code to follow . . . . . . . .>**

1. **}**

## How it works...

As we can see from the example above, that even in 3D the physics collision system has to be divided into phases. The Broad phase collision and the narrow phase. In the Broad phase collision, now we take into Vector3 instead of two float points as we now have three axes (x,y,z). We need to enter the object id and then check within the bounds of the bounding boxes. Similarly, for narrow phase collision, our problem domain and calculations remain same. We now change it to support 3D. The above example shows a part of the problem if we need to find the contact points in the narrow phase collision. We create an array and based on the collision call-backs, we save out all the points which are in contact. Later on we can write other methods to check if the points are overlapping or not.

# Creating a particle system

Particle systems are quite important in games to add to the visual representation of the whole feel of the game. Particle systems are quite easy to write and are merely a collection of one or more particles. So we also need to create a single particle with some properties and then let the particle system decide how many particles it wants.

## Getting ready

1. For this recipe, you will need a Windows machine and an installed version of Visual Studio.

## How to do it...

1. Add a source file called Source.cpp
2. class Particle
3. {
4. Vector3 location;
5. Vector3 velocity;
6. Vector3 acceleration;
7. float lifespan;
8. Particle(Vector3 vec)
9. {
11. acceleration = new Vector3(.05, 0.05);
12. velocity = new Vector3(random(-3, 3), random(-4, 0));
13. location = vec.get();
14. lifespan = 125.0;
15. }
16. void run()
17. {
18. update();
19. display();
20. }
21. void update() {
22. velocity.add(acceleration);
23. location.add(velocity);
24. lifespan -= 2.0;
25. }
26. void display()
27. {
28. stroke(0, lifespan);
29. fill(0, lifespan);
30. trapezoid(location.x, location.y, 8, 8);
31. }
32. boolean isDead()
33. {
34. if (lifespan < 0.0) {
35. return true;
36. }
37. else {
38. return false;
39. }
40. }
41. };
42. Particle p;
43. void setup()
44. {
45. size(800, 600);
46. p = new Particle(new Vector3(width / 2, 10));
47. }
48. void draw()
49. {
50. for (int i = 0; i < particles.size(); i++) {
51. Particle p = particles.get(i);
52. p.run();
53. if (p.isDead()) {
54. particles.remove(i);
55. }
56. }
57. }

## How it works...

As we see in the example, our first task is to create a particle class. The particle class will have properties such as velocity, acceleration, position and life span. Because we are making the particle in 3D space, hence we are using Vector3 to denote the particle properties. If we were to create the particle in 2D space, we could have used Vector2 for the same. In the constructor we assign the starting values of the attributes. We then have two main functions, update and display. The update function updates the velocity and position every frame and also reduced the lifespan so that it disappears when the lifespan is over. In the display function, we need to specify how we want the particle to be viewed. Whether it should have stroke or fill and so on. Over here we also have to specify the shape of the particle. Most common shape is sphere or a cone. We have use trapezoid just to denote that we can specify any shape. Finally, from the client program, we need to call this object and then access the various functions to display the particle.

However, all this will do is display just one particle to the screen. Of course we can create an array of 100 objects and that would display 100 particles to the screen. A better approach is to create a particle system which creates an array of particles. The number of particles that will be drawn is specified by the client program. Based on the request, the particle system draws the required number of particles. Also there must be a function to determine which particles to be removed from the screen. This is dependent on the lifespan of each particle.

# Using Ragdoll in your game

Ragdoll physics is a special kind of procedural animation that is often used as a replacement for traditional static death animations in games. The whole idea between ragdoll animation is that after death, the character falls as if the bones of the body are behaving like a rag. Hence the name. It has nothing to do with realism but adds a special fun element to the game.

## Getting ready

1. For this recipe, you will need a Windows machine and an installed version of Visual Studio. The DirectX SDK is also required.

## How to do it...

#include "RagDoll.h"

#include "C3DETransform.h"

#include "PhysicsFactory.h"

#include "Physics.h"

#include "DebugMemory.h"

RagDoll::RagDoll(C3DESkinnedMesh \* a\_skinnedMesh, C3DESkinnedMeshContainer \* a\_skinnedMeshContainer, int totalParts, int totalConstraints)

{

m\_skinnedMesh = a\_skinnedMesh;

m\_skinnedMeshContainer = a\_skinnedMeshContainer;

m\_totalParts = totalParts;

m\_totalConstraints = totalConstraints;

m\_ragdollBodies = (btRigidBody\*\*)malloc(sizeof(btRigidBody) \* totalParts);

m\_ragdollShapes = (btCollisionShape\*\*)malloc(sizeof(btCollisionShape) \* totalParts);

m\_ragdollConstraints = (btTypedConstraint\*\*)malloc(sizeof(btTypedConstraint) \* totalConstraints);

m\_boneIndicesToFollow = (int\*) malloc(sizeof(int) \* m\_skinnedMesh->GetTotalBones());

m\_totalBones = m\_skinnedMesh->GetTotalBones();

m\_bonesCurrentWorldPosition = (D3DXMATRIX\*\*)malloc(sizeof(D3DXMATRIX) \* m\_totalBones);

m\_boneToPartTransforms = (D3DXMATRIX\*\*)malloc(sizeof(D3DXMATRIX) \* m\_totalBones);

for(int i = 0; i < totalConstraints; i++)

{

m\_ragdollConstraints[i] = NULL;

}

for(int i = 0; i < totalParts; i++)

{

m\_ragdollBodies[i] = NULL;

m\_ragdollShapes[i] = NULL;

}

for(int i = 0; i < m\_totalBones; i++)

{

m\_boneToPartTransforms[i] = NULL;

m\_boneToPartTransforms[i] = new D3DXMATRIX();

m\_bonesCurrentWorldPosition[i] = NULL;

m\_bonesCurrentWorldPosition[i] = new D3DXMATRIX();

}

m\_constraintCount = 0;

}

RagDoll::~RagDoll()

{

free(m\_ragdollConstraints);

free(m\_ragdollBodies);

free(m\_ragdollShapes);

for(int i = 0; i < m\_totalBones; i++)

{

delete m\_boneToPartTransforms[i];

m\_boneToPartTransforms[i] = NULL;

delete m\_bonesCurrentWorldPosition[i];

m\_bonesCurrentWorldPosition[i] = NULL;

}

free(m\_bonesCurrentWorldPosition);

free(m\_boneToPartTransforms);

free(m\_boneIndicesToFollow);

}

int RagDoll::GetTotalParts()

{

return m\_totalParts;

}

int RagDoll::GetTotalConstraints()

{

return m\_totalConstraints;

}

C3DESkinnedMesh \*RagDoll::GetSkinnedMesh()

{

return m\_skinnedMesh;

}

//sets up a part of the ragdoll

//int index = the index number of the part

//int setMeshBoneTransformIndex = the bone index that this part is linked to,

//float offsetX, float offsetY, float offsetZ = translatin offset for the part in bone local space

//float mass = part's mass,

//btCollisionShape \* a\_shape = part's collision shape

void RagDoll::SetPart(int index, int setMeshBoneTransformIndex, float offsetX, float offsetY, float offsetZ,float mass, btCollisionShape \* a\_shape)

{

m\_boneIndicesToFollow[setMeshBoneTransformIndex] = index;

//we set the parts position according to the skinned mesh current position

D3DXMATRIX t\_poseMatrix = m\_skinnedMeshContainer->GetPoseMatrix()[setMeshBoneTransformIndex];

D3DXMATRIX \*t\_boneWorldRestMatrix = m\_skinnedMesh->GetBoneWorldRestMatrix(setMeshBoneTransformIndex);

D3DXMATRIX t\_boneWorldPosition;

D3DXMatrixMultiply(&t\_boneWorldPosition, t\_boneWorldRestMatrix, &t\_poseMatrix);

D3DXVECTOR3 \* t\_head = m\_skinnedMesh->GetBoneHead(setMeshBoneTransformIndex);

D3DXVECTOR3 \* t\_tail = m\_skinnedMesh->GetBoneTail(setMeshBoneTransformIndex);

float tx = t\_tail->x - t\_head->x;

float ty = t\_tail->y - t\_head->y;

float tz = t\_tail->z - t\_head->z;

//part's world matrix

D3DXMATRIX \*t\_partMatrix = new D3DXMATRIX();

\*t\_partMatrix = t\_boneWorldPosition;

D3DXMATRIX \*t\_centerOffset = new D3DXMATRIX();

D3DXMatrixIdentity(t\_centerOffset);

D3DXMatrixTranslation(t\_centerOffset, (tx / 2.0f) + offsetX, (ty / 2.0f) + offsetY, (tz/2.0f) + offsetZ);

D3DXMatrixMultiply(t\_partMatrix, t\_partMatrix, t\_centerOffset);

D3DXVECTOR3 t\_pos;

D3DXVECTOR3 t\_scale;

D3DXQUATERNION t\_rot;

D3DXMatrixDecompose(&t\_scale, &t\_rot, &t\_pos, t\_partMatrix);

btRigidBody\* body = PhysicsFactory::GetInstance()->CreateRigidBody(mass,t\_pos.x, t\_pos.y, t\_pos.z, t\_rot.x, t\_rot.y, t\_rot.z, t\_rot.w, a\_shape);

D3DXMATRIX t\_partInverse;

D3DXMatrixInverse(&t\_partInverse, NULL, t\_partMatrix);

//puts the bone's matrix in part's local space, and store it in m\_boneToPartTransforms

D3DXMatrixMultiply(m\_boneToPartTransforms[setMeshBoneTransformIndex], &t\_boneWorldPosition, &t\_partInverse);

m\_ragdollBodies[index] = body;

delete t\_partMatrix;

t\_partMatrix = NULL;

delete t\_centerOffset;

t\_centerOffset = NULL;

}

//when a bone is not going to have a part directly linked to it, it needs to follow a bone that has

//a part linked to

//int realBoneIndex = the bone that has no part linked

//int followBoneIndex = the bone that has a part linked

void RagDoll::SetBoneRelation(int realBoneIndex, int followBoneIndex)

{

//it is going to the same thing the setPart method does, but the bone it is going to take

//as a reference is the one passed as followBoneIndex and the the part's matrix is below

//by calling GetPartForBoneIndex. Still there is going to be a new entry in m\_boneToPartTransforms

//which is the bone transform in the part's local space

int partToFollowIndex = GetPartForBoneIndex(followBoneIndex);

m\_boneIndicesToFollow[realBoneIndex] = partToFollowIndex;

D3DXMATRIX t\_poseMatrix = m\_skinnedMeshContainer->GetPoseMatrix()[realBoneIndex];

D3DXMATRIX \*t\_boneWorldRestMatrix = m\_skinnedMesh->GetBoneWorldRestMatrix(realBoneIndex);

D3DXMATRIX t\_boneWorldPosition;

D3DXMatrixMultiply(&t\_boneWorldPosition, t\_boneWorldRestMatrix, &t\_poseMatrix);

D3DXMATRIX \*t\_partMatrix = new D3DXMATRIX();

btTransform t\_partTransform = m\_ragdollBodies[partToFollowIndex]->getWorldTransform();

\*t\_partMatrix = BT2DX\_MATRIX(t\_partTransform);

D3DXMATRIX t\_partInverse;

D3DXMatrixInverse(&t\_partInverse, NULL, t\_partMatrix);

D3DXMatrixMultiply(m\_boneToPartTransforms[realBoneIndex], &t\_boneWorldPosition, &t\_partInverse);

delete t\_partMatrix;

t\_partMatrix = NULL;

}

btRigidBody \*\* RagDoll::GetRadollParts()

{

return m\_ragdollBodies;

}

btTypedConstraint \*\*RagDoll::GetConstraints()

{

return m\_ragdollConstraints;

}

void RagDoll::AddConstraint(btTypedConstraint \*a\_constraint)

{

m\_ragdollConstraints[m\_constraintCount] = a\_constraint;

m\_constraintCount++;

}

//This method will return the world position that the given bone should have

D3DXMATRIX \* RagDoll::GetBoneWorldTransform(int boneIndex)

{

//the part world matrix is fetched, and then we apply the bone transform offset to obtain

//the bone's world position

int t\_partIndex = GetPartForBoneIndex(boneIndex);

btTransform t\_transform = m\_ragdollBodies[t\_partIndex]->getWorldTransform();

D3DXMATRIX t\_partMatrix = BT2DX\_MATRIX(t\_transform);

D3DXMatrixIdentity(m\_bonesCurrentWorldPosition[boneIndex]);

D3DXMatrixMultiply(m\_bonesCurrentWorldPosition[boneIndex], m\_boneToPartTransforms[boneIndex], &t\_partMatrix);

return m\_bonesCurrentWorldPosition[boneIndex];

}

int RagDoll::GetPartForBoneIndex(int boneIndex)

{

for(int i = 0; i < m\_totalBones;i ++)

{

if(i == boneIndex)

{

return m\_boneIndicesToFollow[i];

}

}

return -1;

1. }

## How it works...

As you can see from the example above, for this example you would require to have a skinned mesh model. The mesh model can either be downloaded from some royalty free website or made via Blender or any other 3D software packages like Maya or Max. As the whole concept of ragdoll is based on bones of the mesh, we have to make sure that the 3D model has the bones setup correctly.

After that in the code, there are lots of small parts to it. First part of the problem is to write a bone container class which stores all the bone information. Next in our ragdoll class, we need to use the bone container class and by using the Bullet physics SDK, assign rigid body to each of the bones. After the rigid body has been setup, we need to traverse through the bones once again and create a relationship between each bone. So that when one bone moves, the neighbouring bones moves in accordance to it. Finally we also need to add the constraints so that when the physics engine simulates the ragdoll, it can solver the constraints properly and output the result to the bones.