PART 18: “Physics”

Introduction

In this tutorial, you will be introduced to game physics. This is an incredibly broad field, and quite complex too, so you’ll really only get a teaser. This will however produce some pretty effects.

Physics

Many recent games use highly advanced *physics engines*. A physics engine is a piece of software that gets linked to a game. It takes care of natural behavior of objects in the game, taking into account forces, collisions, momentum, rotation and torque. Writing a good physics engine is hard for a number of reasons:

Some game objects move fast, which makes accurate calculations tricky;
Simulation in a game happens in steps, rather than continuously, like in nature.

For this tutorial, we will therefore focus on a simplified physics model, called *Verlet Integration*.

Verlet: basic idea

It works like this. If you have an object that moves along a straight line at a fixed speed, for each frame of your game, you can calculate its position like this:

```
position = origin + speed * traveltime
```

That’s one way. Another way just looks at subsequent frames, and calculates the position incrementally:

```
position += speed * frametime
```

If the time it takes to render one frame is roughly the same each time, you can simplify this one step further:

```
position += (position – previous position)
```

In other words: if we moved from A to B during the last frame, we will move the same distance for the next frame, if frame time and speed remains the same. The distance between two ‘snapshots’ now effectively becomes (directional) speed.

Now imagine that something is pulling on our object, e.g. gravity. To do that, we adjust the position (by adding `position – previous position`) and then we move the object some more, to account for this force. If the object was already moving down, it will now move down faster, because in the next frame, `position – previous position` includes the applied gravity.

Let's put this into some code. First, I need a piece of code to draw a circle. That's because circles (and spheres, in 3D) are excellent for physics. You'll soon enough find out why.

```
void Circle( Surface* s, float x, float y, float r )
{
    for ( int i = 0; i < 64; i++ )
    {
        float r1 = (float)i * PI / 32, r2 = (float)(i + 1) * PI / 32;
        s->Line( x - r * sinf( r1 ), y - r * cosf( r1 ),
                 x - r * sinf( r2 ), y - r * cosf( r2 ), 0xff0000 );
    }
}
```

No need to scrutinize that code, for now it's fine to simply use it.
Then, the falling ball code, using Verlet integration:

```
float x[2] = { 300, 320 }, y[2] = { 0, SCRHEIGHT - 50 };
float px[2] = { 300, 320 }, py[2] = { 0, SCRHEIGHT - 50 };
void Game::Tick( float deltaTime )
{
       screen->Clear( 0 );
       Circle( screen, x[0], y[0], 16 );
       Circle( screen, x[1], y[1], 49 );
       // backup current position
       float lx = x[0], ly = y[0];
       // verlet integration
       x[0] += (x[0] - px[0]);
       y[0] += (y[0] - py[0]);
       // forces
       y[0] += 0.5f;
       // store new px and py
       px[0] = lx, py[0] = ly;
       // delay
       Sleep( 50 );
}
```

As you can see, the ball starts falling, and increases speed. It does fall through the floor though… And let's not forget the other sphere.

Verlet: constraints

So, we now have subsequent positions (the difference of which is speed), and forces (implemented by changing positions directly). We need one more thing: *constraints*. In nature, a ball cannot fall through another ball, or through the floor. Using Verlet integration, we handle this using constraints.

Here's one example: since a ball cannot fall through the floor, we need to test for that. As soon as we detect that a ball *does* intersect the floor, we correct the situation, by moving the ball out of the floor. Note that this effectively changes the speed of the ball, if we keep interpreting the difference between subsequent positions as speed!

Apart from avoiding collisions using constraints, we can also keep objects together, using simple springs. This time, the constraint is: two objects may not be further apart

than a certain distance. When they are, we move both objects closer together to simulate the spring.

So, let's add a first constraint. Here's the code:

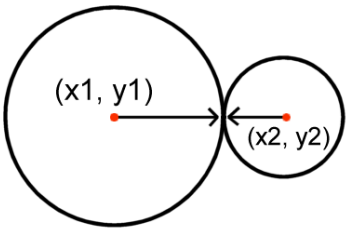
```
// constraints
if (y[0] > (SCRHEIGHT - 17)) y[0] = SCRHEIGHT - 17;
```

Add this code to the Tick function; insert it right after the 'forces' bit.
The result may be a bit unsatisfying. No bouncing? We can improve it a little bit:

```
if (y[0] > (SCRHEIGHT - 17)) ly = y[0], y[0] = SCRHEIGHT - 17;
```

But that's a hack. In general, Verlet integration is not very good for bounces, sadly. Let's see what it is good at: the other sphere.

How can be determine if the two balls collide? We need some math for that. If ball 1 is located at (x1, y1), and ball 2 is located at (x2, y2), then the distance is the square root of $(x2 – x1)^2 + (y2 – y1)^2$. If this distance drops below the combined sizes of the balls, they intersect.



When the circles intersect, we have a number of options.

1. We could move the smallest ball, so that it doesn't intersect anymore;
2. We could move both balls;
3. We could move both balls, but the lightest ball more than the heavier one.

Option 3 is clearly the most realistic one. But how do we move the balls away from each other? Maths to the rescue again: if the circles intersect, their distance is smaller than their combined radii. The combined radii minus their distance is the penetration depth, and thus the distance they need to be moved apart. Now, if the small ball is ten times lighter than the large one, the small one will do 90% of this distance, and the big one only 10%. Here we go:

```
float xdist = x[1] - x[0];
float ydist = y[1] - y[0];
float distance = sqrtf( xdist * xdist + ydist * ydist );
if (distance < (16 + 49))
{
    float fix = (16 + 49) - distance;
    float fraction = fix / distance;
    x[0] -= fraction * xdist;
    y[0] -= fraction * ydist;
}
```

In words: for two balls with radius 16 and 49, the distance should be at least 16 + 49. If it isn't, then they intersect, by (16 + 49) – distance. They need to be moved apart, and not just in any direction, but along the line that connects their centers.

Once that is done, the small ball responds surprisingly realistic to the collision.

Assignment

There is clearly a very large amount of experiments that you could carry out with this code, so today's assignment is a somewhat random selection.

1. Fix the code so that the big ball moves. Assume that it is heavier than the small ball.
2. Add vertical walls, so that the small ball doesn't leave the screen anymore.
3. Add some extra small balls. Make sure that *all* balls collide with *all* other balls. Do not forget that ball x should not collide with ball x (i.e., it should skip itself in all calculations).
4. Add a spring: connect a heavy ball to a fixed point, and if it gets further than a preset distance from that point, pull it towards the point using Verlet physics.

Let it rain… At some point, you will bring C/C++ to its knees. How to make code like this faster is the topic for another day.

END OF PART 16

Next part: "AI"