Project N.G.P.

Team 52

Jamal Osman
Josh Rice
Joey Helgeson

Table of Contents

Executive Summary

In this project we are making a 3d game using the engine 3dgamestudio and C-script. This game will demonstrate newtonian physics and realistic gameplay. We hope to simulate the different factors of physics that are in real life. Also we will incorporate realistic movement in the sprites. We hope to progress throughout the year and make this a worthwhile project.

Introduction

The impressive capabilities of the latest generation of video game hardware have raised our expectations of not only how digital characters look, but also how they move. As a result, game developers are becoming increasingly interested in animation techniques to generate natural looking motion for the complex characters that populate the modern game worlds. These characters
should be able to respond realistically to unpredictable user control and interact with their environments in a believable way, requirements that traditional hand animation and motion capture fail to adequately address. Developers are beginning to look at physics as a way to address some of the shortcomings of keyframed motion. Until recently, physics was considered an off-line process, but faster hardware and improved algorithms have helped change that perception. As a matter of fact, it is quite common these days for games to use real-time dynamics solvers for particles, cloth, soft and rigid bodies.

Significance

In this paper we will answer all these questions, but in order to keep this paper focused we must concentrate on the simulation of just a single type of object. The class of objects we will study are called a *rigid bodies*. Rigid meaning the objects cannot bend, compress or deform in any way. This paper explains rigid body *kinematics* and *dynamics*. Kinematics is the study of how an object moves in the absence of forces, while dynamics describes how an object reacts to them. Together kinematics and dynamics provide all the information you need to simulate the motion of an object. Along the way I will show you how to integrate vector quantities, handle rotations in three dimensions and display your object as it moves and spins around the world.

Procedure

As long as we only have single floating point values for position and velocity, our physics simulation will be limited to motion in a single dimension, and a point moving from side to side on the screen is pretty boring. If we want our simulation to move in three dimensions, it makes sense that we need to track its motion separately in each dimension: left and right, forward and back, up and down. If we apply the equations of motion to each dimension separately, then we can integrate each dimension in turn to find the motion of the object in three dimensions.

Or we could just use vectors.

Vectors are a mathematical type representing an array of numbers. In three dimensions, vectors have three components x, y and z. Each component corresponds to a dimension. In this project x is left and right, y is up and down, and z is forward and back.

In C-script we will implement vectors using a struct as follows:

```
    struct Vector
    {
         float x,y,z;
    };
```

Addition of two vectors is defined as simply adding each component together, and multiplying a vector by a floating point number is the same as just multiplying each component. Lets add overloaded operators to the vector struct so that we can perform these operations in code as if vectors are a native type:

```
    struct Vector
    {
         float x,y,z;

         Vector operator+(const Vector &other)
         {
              Vector result;
              result.x = x + other.x;
```

```
            result.y = y + other.y;
            result.z = z + other.z;
            return result;
        }

        Vector operator*(float scalar)
        {
            Vector result;
            result.x = x * scalar;
            result.y = y * scalar;
            result.z = z * scalar;
            return result;
        }
    };
```

Now, instead of maintaining completely seperate equations of motion and integrating seperately for x, y and z, we will convert our position, velocity, acceleration and force to vector quantities, then integrate the vectors directly using the equations of motion:

F = m**a** (force equals mass times acceleration)
d**v**/dt = **a** (the derivative of velocity is acceleration)
d**x**/dt = **v** (the derivative of position is velocity)

Notice how **F**, **a**, **v** and **x** are written in bold. This is the convention used to distinguish vector quantities from scalar (single value) quantities such as mass m and time t.

Now that we have the equations of motion in vector form, how do we integrate vector quantities? The answer is exactly the same as we integrated single values. This is because we added overloaded operators for adding two vectors together, and multiplying a vector by a scalar. This is all we need to be able to drop in vectors in place of floats and have everything just work, even with the complex RK4 integrator.

For example, here is a simple Euler integration for vector position from velocity:

```
    position = position + velocity*dt;
```

Notice how the overloaded operators make it look exactly the same as an Euler integration for a single value. But what is it really doing? Lets take a look at how we would implement vector integration *without* the overloaded operators:

```
    position.x = position.x + velocity.x * dt;
    position.y = position.y + velocity.y * dt;
    position.z = position.z + velocity.z * dt;
```

Its exactly the same as if we integrated each component of the vector separately. This is the cool thing about vectors. Whether we integrate vectors directly, or integrate each component seperately, we are doing *exactly the same thing.*

From now on every object will have its own mass in kilograms so the simulation needs be driven by forces instead. There are two ways we can do this. Firstly, we can divide force by mass to get acceleration as we are used to, then integrate this acceleration to get the velocity, and integrate velocity to get position. The second way is to integrate force directly to get momentum, then convert this momentum to velocity by dividing it by mass, and finally integrate velocity to get position. Remember that momentum is simply velocity multiplied by mass:

d**p**/dt = **F** (the derivative of momentum is force)
v = **p**/m (velocity equals momentum divided by mass)
d**x**/dt = **v** (the derivative of position is velocity)

Both methods work, but in my opinion the second way is better because it is consistent with the way that we must approach rotation. When we switch to the second technique, it adds a new level of complexity to our code. Each time that momentum changes we need to make sure that the velocity is recalculated by dividing momentum by mass. Doing this manually everywhere that momentum is changed would be error prone and a chore. So we now separate all our state quantities into *primary*, *secondary* and *constant* values, and add a method called 'recalculate' to the State struct which is responsible for updating all the secondary values from the primary ones:

```
    struct State
    {
         // primary
         Vector position;
         Vector momentum;

         // secondary
         Vector velocity;

         // constant
         float mass;
         float inverseMass;

         void recalculate()
         {
              velocity = momentum * inverseMass;
         }
    };

    struct Derivative
    {
         Vector velocity;
         Vector force;
    };
```

If we make sure that recalculate is called whenever any of the primary values change, then our secondary values will always stay in sync. This is important because we need to use the secondary value velocity when integrating position, but we integrate momentum directly from force. This may seem like overkill just to handle converting momentum to velocity, but as our simulation becomes more complex we will have many more secondary values, so its important to design a system that scales well.

Finally, a minor point. Notice how I store inverseMass (1.0/mass) as well as mass and multiply by the inverse of mass when converting momentum to velocity instead of dividing by mass directly. This is because floating point multiplication is significantly faster than division. It is good practice to do this whenever you can in your physics simulation.

So far we have covered linear motion, that is, we can simulate an object so that it can move in 3D space and have forces applied to it, but it *cannot rotate*. The good news is that rotational equivalents to force, momentum, velocity, position and mass exist, and once we understand how they work, integration of rotational physics state can be performed using the RK4 integrator and our object will spin.

So lets start off by talking about how an object rotates. The bodies that we are simulating are rigid meaning that they cannot deform. The key point to get out of this is that when moving freely, a rigid body will move with both a linear component (position, velocity, momentum etc.) which we have already covered, and a rotational component rotating about its *center of mass*.

The center of mass of the object is the weighted average of all points making up the body by their mass. For objects of uniform density, the center of mass is always the geometric center of the object, for example the center of a sphere or a cube.

So how do we represent how the object is rotating? If you think about it you'll realize that rotation can only ever be around a single axis at any time, so the first thing we need to know is what that axis is. We can represent this axis with a unit length vector. Next we need to know how fast the object is rotating about this axis in radians per second.

If we know the center of mass of the object, the axis of rotation, and the speed of rotation then we have complete information about how the object is rotating.

The standard way of representing rotation over time is by combining the axis and the speed of rotation into a single vector called *angular velocity*. This means that the length of the angular velocity vector is the speed of rotation in radians while the direction of the vector indicates the axis of rotation. For example, an angular velocity of (2Pi,0,0) indicates a rotation about the x axis doing one revolution per second.

But what direction is this rotation in? In the example source code I use a right handed coordinate system which is standard when using OpenGL. To find the direction of rotation just take your *right hand* and point your thumb down the axis - your fingers will now curl in the direction of rotation. If your 3D engine uses a left handed coordinate system then just use your left hand instead.

Why do we combine the axis and rate of rotation into a single vector? Doing so gives us a single vector quantity that is easy to manipulate just like velocity for linear motion. We can easily add and subtract changes to angular velocity to change how the object is rotating just like we can add and subtract from linear velocity. If we stuck with a unit length vector and scalar for rotation speed then it would be much more complicated to apply changes to angular velocity.

But there is one very important different between linear and angular velocity. Unlike linear velocity, there is no guarantee that angular velocity will remain constant over time in the absence of forces. In other words, angular momentum is conserved while angular velocity is not. This means that we cannot trust angular velocity as a primary value and we need to use angular momentum instead. This is the reason we switched to integrating momentum from force in the previous section.

Just as velocity and momentum are related by mass in linear motion, angular velocity and angular momentum are related by a quantity called *inertia*. Inertia is a measurement of how much effort it takes to spin an object around an axis, and it is a property of how much the object weighs *and* the shape of the object.

Generally inertia is represented as a tensor which in 3D physics ends up as a 3x3 matrix. However, We will discuss physics in the context of the simulation of a cube. And because of the symmetries of the cube, we only need a single value: $inertia = 1/6 \times size^2 \times mass$, where size is the length of the sides of the cube. For now lets consider this single inertia value as the rotational equivalent of mass.

Just as we integrate linear momentum from force, we integrate angular momentum directly from the rotational equivalent of force called *torque*. You should think of torque just like a force, except that when it is applied it induces a rotation around an axis in the direction of torque vector rather than accelerating the object linearly. For example, a torque of (1,0,0) will cause a stationary object to rotate about the x axis.

Once we have angular momentum integrated, we multiply it by the inverse of the inertia to get the angular velocity, and using this angular velocity we integrate to get the rotational equivalent of position called *orientation*. However, as we will see, integrating orientation from angular velocity is a bit more complicated.

This complexity is due to the difficulty of representing orientations in three dimensions.

In two dimensions orientations are easy, you just keep track of the angle in radians and you are done. In three dimensions it becomes much more complex. It turns out that you must either use 3x3 rotation matrices or quaternions to correctly represent the orientation of an object.

For reasons of simplicity and efficiency I'm going to use quaternions to represent the orientation instead of matrices. This also gives us an easy way to interpolate between the previous and current physics orientation to get smooth framerate independent animation as per the time stepping scheme.

Now there are plenty of resources on the internet which explain what quaternions are and how unit length quaternions are used to represent rotations in three dimensions. Here is a particularly nice one http://www.sjbrown.co.uk/quaternions.html. What you need to know however is that, *effectively,* unit quaternions represent an axis of rotation and an amount of rotation about that axis, similar to how angular velocity represents rotation about an axis, but in a mathematically different form.

We will represent quaternions in code as another struct:

```
    struct Quaternion
    {
         float w,x,y,z;               // overloaded
operators etc. omitted
    };
```

If we define the rotation of a quaternion as being relative to an initial orientation of the object (what we will later call *body coordinates*) then we can use this quaternion to represent the orientation of the object at any point in time. Now that we have decided on the representation to use for orientation, we need to integrate it over time so that the object rotates according to the angular velocity.

We are now presented with a problem. Orientation is a quaternion but angular velocity is a vector. How can we integrate orientation from angular velocity when the two quantities are in different mathematical forms?

The solution is to convert angular velocity into a quaternion form, then to use this quaternion to integrate orientation. For lack of a better term I will call this time derivative of the orientation quaternion *spin.* Exactly how to calculate this spin quaternion is detailed in http://www-2.cs.cmu.edu/~baraff/sigcourse/notesd1.pdf.

Here is the final result:

d***q***/dt = **spin** = 0.5 ***w q***

Where ***q*** is the current orientation quaternion, and ***w*** is the current angular velocity in quaternion form (0,x,y,z) such that x, y, z are the components of the angular velocity vector. Note that the multiplication done between ***w*** and ***q*** is quaternion multiplication.

To implement this in code we add spin as a new secondary quantity calculated from angular velocity in the recalculate method. We also add spin to the derivatives struct as it is the derivative of orientation:

```
    struct State
    {
         // primary
         Quaternion orientation;
         Vector angularMomentum;

         // secondary
         Quaternion spin;
         Vector angularVelocity;

         // constant
         float inertia;
         float inverseInertia;

         void recalculate()
         {
              angularVelocity = angularMomentum *
inverseInertia;
```

```
              orientation.normalize();

              spin = 0.5f * Quaternion(0, 
angularVelocity.x, angularVelocity.y, angularVelocity.z) * 
orientation;
         }
    };

    struct Derivatives
    {
         Quaternion spin;
         Vector torque;
    };
```

Integrating a quaternion, just like integrating a vector, is as simple as doing the integration for each value separately. The only difference is that after integrating orientation we must renormalize the orientation quaternion (make it unit length) in order to ensure that it still represents a rotation. This is because errors in integration accumulate over time and make the quaternion 'drift' away from being unit length. I like to do the renormalization in the recalculate method for simplicity, but you can get away with doing it much less frequently if cpu cycles are tight.

Now in order to drive the rotation of the object, we need a method that can calculate the torque applied given the current rotational state and time just like the force method we use when integrating linear motion. eg:

```
    Vector torque(const State &state, float t)
    {
         return Vector(1,0,0) - state.angularVelocity * 
0.1f;
    }
```

This function returns an acceleration torque to induce a spin around the x axis, but also applies a damping over time so that at a certain speed the accelerating and damping will cancel each other out. This is done so that the rotation will reach a certain rate and stay constant instead of getting faster and faster over time.

Now that we are able to integrate linear and rotational effects, how can they be combined into one simulation? The answer is to literally integrate both the linear and rotational physics state simultaneously and everything works out. This is because the objects we are simulating are rigid so we can decompose their motion into separate linear and rotational components. As far as integration is concerned, you can treat linear and angular effects as being completely independent of each other.

Now that we have an object that is translating and rotating through three dimensional space, we need a way to keep track of where it is. We must now introduce the concepts of *body coordinates* and *world coordinates*.

Think of body coordinates in terms of the object in a convenient layout, for example its center of mass would be at the origin (0,0,0) and it would be oriented in the simplest way possible. In the case of the simulation, in body space the cube is oriented so that it lines up with the x, y and z axes and the center of the cube is at the origin.

The important thing to understand is that the object remains stationary in body space, and is transformed into world space using a combination of translation and rotation operations which put it in the correct position and orientation for rendering. When you see the cube animating on screen it is because it is being drawn in world space using the body to world transformation.

We have the raw materials to implement this transform from body coordinates into world coordinates in the position vector and the orientation quaternion. The trick to combining the two is to convert each of them into 4x4 matrix form which is capable of representing both rotation and translation. Then we combine the two transformations into a single matrix by multiplication. This combined matrix has the effect of first rotating the cube around the origin to get the correct orientation, then translating the cube to the correct position in world space. See http://www.gamedev.net/reference/articles/article695.asp for details on how this is done.

If we then invert this matrix we get one that has the opposite effect, it transforms points in world coordinates into the body coordinates of the object. Once we have both these matrices we have the ability to convert points from body to world coordinates and back again which is very handy. These two matrices become new secondary values calculated in the 'recalculate' method from the orientation quaternion and position vector.

We can apply separate forces and torques to an object individually, but we know from real life that if we push an object it usually makes it both move and rotate. So how can we break down a force applied at a point on the object into linear force which causes a change in momentum, and a torque which changes angular momentum?

Given that our object is a rigid body, what actually happens here is that the entire force applied at the point is applied linearly, plus a torque is also generated based on the cross product of the force vector and the point on the object relative to the center of mass of the object:

$$\mathbf{F}_{\text{linear}} = \mathbf{F}$$
$$\mathbf{F}_{\text{torque}} = \mathbf{F} \times (\mathbf{p} - \mathbf{x})$$

Where **F** is the force being applied at point **p** in world coordinates, and **x** is the center of mass of the object. This seems counterintuitive at first. Is the force being applied twice accidentally? In fact this is our everyday experience with objects clouding the true behavior of an object under ideal conditions.

Consider a bowling ball lying on a slippery surface such as ice so that no significant friction is present. Now in your mind try to work out a way that you can apply a force at a single point on the surface of the bowling ball such that it will stay completely still while rotating on the spot. In truth there is no way to do this. Whatever force we manage to apply to the bowling ball, there will always be a linear force component which causes the ball to accelerate.

The only way that we can apply a force at a single point which makes a rigid body only rotate is if the object is constrained to rotate but not move. For example, if we apply a force to the tire of a bicycle wheel lifted off the ground it appears to result in only a torque, but this is only because the axle of the wheel constrains the wheel's motion in such a way as to counteract the linear component of the force we applied. Consider what would happen to the wheel if you were riding the bicycle and the axle disappeared.

The final piece of the puzzle is how to calculate the velocity of a single point in the rigid body. To do this we start with the linear velocity of the object, because all points must move with this velocity to keep it rigid, then add the velocity at the point due to rotation.

This velocity due to rotation will not be constant for every point in the body if it is rotating, as each point in the body must be spinning around the axis of rotation. Combining the linear and angular velocities, the total velocity of a point in the rigid body is:

$$\mathbf{v}_{point} = \mathbf{v}_{linear} + \mathbf{v}_{angular} * (\mathbf{p} - \mathbf{x})$$

Where **p** is the point on the rigid body and **x** is the center of mass of the object. Knowing the velocity of a point on the body is important for many calculations such as collision detection and response.

Conclusion

We have covered the techniques required to simulate linear and rotational movement of a rigid body in three dimensions. By combining the linear and rotational physics into a single physics state and integrating, we can simulate the motion of a cube in three dimensions as it moves and spins around.

Bibliography

http://www.gamedev.net/reference/articles/article695.asp
http://www.sjbrown.co.uk/quaternions.html
Game Physics, First Edition (The Morgan Kaufmann Series in Interactive 3D Technology)
www.devmaster.net
Physics for Game Developers
www.terminal26.de

Source Code

```
////////////////////////////////////////////////////////////////////////////////
// Roller main script
/////////////////////////////////////////////////////////////////////////
// Files to over-ride:
// * logodark.bmp - the engine logo, include your game title
// * horizon.pcx - A horizon map displayed over the sky and cloud maps
/////////////////////////////////////////////////////////////////////////
```

```
// The PATH keyword gives directories where game files can be found,
// relative to the level directory
path "C:\\Program Files\\GStudio6\\template";          // Path to WDL templates
subdirectory

/////////////////////////////////////////////////////////////////////////////


/////////////////////////////////////////////////////////////////////////////
// The engine starts in the resolution given by the follwing vars.
var video_mode = 8;    // screen size 640x480
var video_depth = 16; // 16 bit colour D3D mode

/////////////////////////////////////////////////////////////////
// Strings and filenames
// change this string to your own starting mission message.

string level_str = <roller.WMB>; // give file names in angular brackets

/////////////////////////////////////////////////////////////////
// define a splash screen with the required A4/A5 logo
bmap splashmap = <logodark.pcx>; // the default logo in templates
panel splashscreen {
        bmap = splashmap;
        flags = refresh,d3d;
}

/////////////////////////////////////////////////////////////////////////////
// The following script controls the sky


/////////////////////////////////////////////////////////////////
// The main() function is started at game start
font digit_font=<digfont.pcx>,12,16;bmap rllr=<roller.bmp>;bmap
rllr2=<roller2.bmp>;bmap pointmap=<gb4.bmp>;
var point;var balls; var pointballs;var phys=1; var play;FONT
digit_font2=<font1_blue.pcx>,12,16;
panel intro
{bmap=rllr;
pos_y=0;pos_x=0;
}
panel end
{bmap=rllr2;
pos_y=0;pos_x=0;
digits 200,270,2,digit_font,10,point;
digits 330,270,2,digit_font,10,balls;
digits 220,320,2,digit_font2,1,pointballs;}
panel points
{bmap=pointmap;
pos_y=0;pos_x=0;
digits 10,10,2,digit_font,10,point;
flags visible,refresh;}
function main()
{SKY_CLIP=-90;
// set some common flags and variables
//	warn_level = 2;		// announce bad texture sizes and bad wdl code
	tex_share = on;		// map entities share their textures

// center the splash screen for non-640x480 resolutions, and display it
	splashscreen.pos_x = (screen_size.x - bmap_width(splashmap))/2;
	splashscreen.pos_y = (screen_size.y - bmap_height(splashmap))/2;
	splashscreen.visible = on;
// wait 3 frames (for triple buffering) until it is flipped to the foreground
	wait(3);

// now load the level
	level_load(level_str);
// freeze the game
	freeze_mode = 1;

// wait the required second, then switch the splashscreen off.
	sleep(1);
	splashscreen.visible = off;
	bmap_purge(splashmap);	// remove splashscreen from video memory

// load some global variables, like sound volume


// display the initial message


// initialize lens flares when edition supports flares


// use the new 3rd person camera


// un-freeze the game
	freeze_mode = 0;

//	client_move();	// for a possible multiplayer game
// call further functions here...
```

```
intro.pos_x=(screen_size.x - bmap_width(rllr))/2;
intro.pos_y= (screen_size.y - bmap_height(rllr))/2;
intro.visible=on;
sleep(5);intro.visible=off;play=1;end.pos_x=(screen_size.x - bmap_width(rllr))/2;
end.pos_y= (screen_size.y - bmap_height(rllr))/2;

}


//////////////////////////////////////////////////////////////////
// The following definitions are for the pro edition window composer
// to define the start and exit window of the application.
WINDOW WINSTART
{
        TITLE           "3D GameStudio";
        SIZE            480,320;
        MODE            IMAGE;      //STANDARD;
        BG_COLOR        RGB(240,240,240);
        FRAME                   FTYP1,0,0,480,320;
//      BUTTON
        BUTTON_START,SYS_DEFAULT,"Start",400,288,72,24;
        BUTTON          BUTTON_QUIT,SYS_DEFAULT,"Abort",400,288,72,24;
        TEXT_STDOUT     "Arial",RGB(0,0,0),10,10,460,280;
}

/* no exit window at all..
WINDOW WINEND
{
        TITLE           "Finished";
        SIZE            540,320;
        MODE            STANDARD;
        BG_COLOR        RGB(0,0,0);
        TEXT_STDOUT     "",RGB(255,40,40),10,20,520,270;

        SET FONT        "",RGB(0,255,255);
        TEXT            "Any key to exit",10,270;
}*/


//////////////////////////////////////////////////////////////////
//INCLUDE <debug.wdl>;
var place[3];var place2[3];var camrot=0;var camp;entity* ballcam;
action baller
{ballcam=me;
}
function camera1
```

```
{while(1){if(camp==1){while(camera.x>my.x-100){camera.x-=1*time;camera.y = my.y;
        camera.z = my.z + 80;
vec_set(temp.x,my.x);vec_sub(temp.x,camera.x);vec_to_angle(camera.pan,temp.x);
wait(1);} camp=0;


}if(camp==0){camera.x = my.x-100;
        camera.y = my.y;
        camera.z = my.z + 80;
vec_set(temp.x,my.x);vec_sub(temp.x,camera.x);vec_to_angle(camera.pan,temp.x);

}if(camp==2){camera.x = my.x-10;
        camera.y = my.y;
        camera.z = my.z + 80;
camera.pan+vec_set(temp.x,my.x);vec_sub(temp.x,camera.x);vec_to_angle(camera.pan,t
emp.x);

}if(camp==3){camera.x = my.x-1;
        camera.y = my.y;
        camera.z = my.z - 80;
vec_set(temp.x,my.x);vec_sub(temp.x,camera.x);vec_to_angle(camera.pan,temp.x);

}if(camp==4){
       camera.x=my.x+10;camera.y = my.y;camera.z = my.z ;


}if(camp==5){camera.x = my.x+80;
        camera.y = my.y;
        camera.z = my.z + 80;
vec_set(temp.x,my.x);vec_sub(temp.x,camera.x);vec_to_angle(camera.pan,temp.x);

}
wait(1);}
}var d1=0;
function reset
{while(my.z>-1129){wait(1);
}phent_settype(my,0,0);my=null;level_load(level_str);d1=0;camp=0;point=0;balls=0;poi
ntballs=0;
}
action roller
{camera1();reset();phys=1;player=me;wait(10);ballcam=me;my.enable_impact=on;my.e
nable_block=on;my.enable_trigger=on;my.trigger_range=10;phent_settype(my,ph_rigid,
ph_sphere);phent_setmass(my,1,ph_sphere);
temp.x=0;temp.y=0;temp.z=-386;
ph_setgravity(temp);phent_setfriction(my,30);phent_setelasticity(my,0,10);
while(my!=null){if(phys==1){if(play==1){
```

```
if(key_a==1){phent_addcentralforce(my,vector(0,200,0));}
if(key_d==1){phent_addcentralforce(my,vector(0,-200,0));}
if(key_w==1){phent_addcentralforce(my,vector(200,0,0));}
if(key_s==1){phent_addcentralforce(my,vector(-200,0,0));}
}}if(phys==0){phent_settype(my,0,0);break;
}
wait(1);
}
}

function door1
{if(event_type==event_impact&&you.z>my.z+25){while(my.z>67){my.z-=.1*time;
wait(1);}
}d1=1;
}
action s1
{my.enable_impact=on;my.event=door1;
}

action gate1
{while(my.z>-32){if(d1==1){my.z-=2*time;
}wait(1);
}
}
function flipcam()
{if(event_type==event_impact){wait(100);camp=2;
}
}
action camch
{my.enable_impact=on;my.trigger_range=100;my.event=flipcam;
}
function flipcam2()
{if(event_type==event_impact){camp=1;my.enable_impact=off;
}
}
action camch2
{my.enable_impact=on;my.transparent=on;my.alpha=70;my.blue=200;my.red=1;my.gre
en=1;my.light=on;my.trigger_range=100;my.event=flipcam2;
while(1){my.v+=5;wait(1);
}
}

function goup()
{if(event_type==event_impact){camrot=0;place2.x=ballcam.x;place2.y=ballcam.y;my.en
able_impact=off;while(camrot<30){camp=0;wait(1);camrot+=.1;
```

```
phent_addcentralforce(ballcam,vector(0,0,500));}my.enable_impact=on;camp=0;}
}
action jumpup
{my.enable_impact=on;wait(10);my.event=goup;
}
action btube
{my.transparent=on;my.alpha=60;while(1){my.u+=1*time;wait(1);
}
}
function warp()
{if(event_type==event_impact){phys=0;pointballs=(point/balls)*100;
end.visible=on;
}}
bmap sparkle=<strall.pcx>;
function smoke_property()
{my.alpha-=10*time;MY.SIZE-=2*TIME;
if (my.alpha<=00) {
	my.lifespan=0;
}}
function fire1()
{my.size=20;my.bmap=sparkle;my.move=on;my.bright=on;my.vel_z=random(10)-
5;my.vel_x=random(10)-5;my.vel_y=random(10)-
5;MY.RED=1;MY.GREEN=255;MY.BLUE=100;
my.flare=on;my.alpha=100;my.function=smoke_property;}
action warps
{my.enable_impact=on;wait(10);my.event=warp;while(1){effect(fire1,1,my.x,normal);w
ait(1);
}
}string redb2=<PART_missile_trail.PCX>;
	FUNCTION EXPLOFLARE2
{MY.PASSABLE=ON;my.oriented = off;
my.facing = on;
MY.SCALE_X=.001;MY.SCALE_Y=.001;MY.SCALE_Z=.001;VEC_TO_ANGLE(M
Y.PAN,NORMAL);
my.alpha=100;MY.TRANSPARENT=ON;my.bright=on;MY.RED=RANDOM(255);
MY.BLUE=RANDOM(255);
MY.GREEN=RANDOM(255);MY.LIGHT=ON;my.flare=on;MY.TRANSPARENT=O
N;while(MY.ALPHA >= 0)
	{
		MY.SCALE_X += 3*TIME;MY.SCALE_Y += 3*TIME;MY.SCALE_Z
+= 3*TIME;MY.ALPHA-=2*TIME;
		WAIT(1);
	}remove(my);}
function collect
```

```
{while(vec_dist(my.x,player.x)>30){wait(1);}media_play("bounce.wav",null,100);point+
=1;my.passable=on;my.red=255;my.blue=255;my.green=255;my.light=on;ENT_CREAT
E(REDB2,MY.X,EXPLOFLARE2);remove(my);
}

action cllct
{balls+=1;my.passable=on;collect();
}

action cam_switcher
{camp==0;while(1){wait(1);my.passable=on;my.invisible=on;if(vec_dist(my.x,player.x)
<10){camp=4;
}
}
}
```

Screenshots