Cubemap based collision detection

Xavier Bourry[1]*

Abstract
We provide an algorithm for character-scenery physics. Instead of using CPU geometry collision dectection or GPU special computing units, we use an algorithm running mainly on GPU, based on cubemapping. It do not require any advanced functionnality of the GPU and can be implemented on mobile devices with OpenGL ES or with WebGL. This algorithm provides a significant performance increase at loading and at running of the application over CPU based algorithm.

Keywords
cubemapping — collision — detection — cubemap — GPU — shaders — octree — WebGL — OpenGL — character — scenery

[1] *SPACEGOO - http://www.spacegoo.com*

Contents

Introduction 1

1 Principle 1

1.1 Collision detection 1
The cubemap • The projection • The vertex shader • The fragment shader

1.2 Path correction 2
Computing collision vectors • Get max values • Displacement of the sphere

2 Implementation 3

2.1 The drawing loop 3

2.2 WebGL 3

References 3

Introduction

Physics is an important aspect of 3D applications, to avoid collision between the character and the scenery. It is also often computing intensive.

The usual algorithm requires to compute an octree for the sscenery meshes. Then collisions between the character and the scenery are computed using sphere-octree collision detection algorithm. [1] [2] [3] The octree can be either precomputed and included into meshes data, or computed at the loading of the application. In both cases, using an octree increase either data volume, either loading duration. On modern gaming graphic hardwares, there are units specialized in physical computing, like PhysX for Nvidia Geforce. But they are unavailable on low-end graphic devices.

Our algorithm computes physics by rendering a world axis aligned depth cubemap. It can work with low-end graphic devices, and computation are done mainly on GPU.

1. Principle

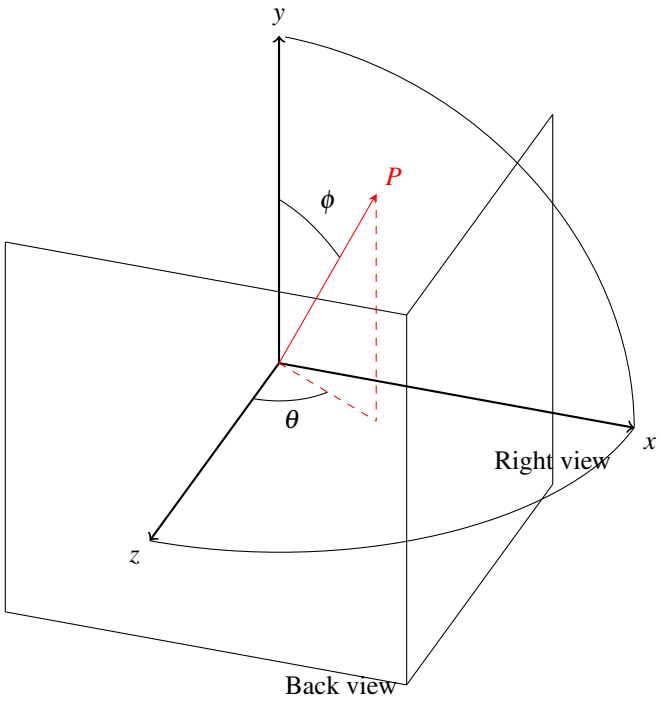

Figure 1. Only back and right projection planes are shown

1.1 Collision detection

The first step is to detect collisions between the character sphere and the scenery. The character sphere is approximatively the bounding box of the character mesh. Our algorithm can work as well for first person view as for third person view.

1.1.1 The cubemap

In the rendering loop, before rendering the scene, we render a depth cubemap. We use the usual Cartesian coordinate system of the 3D programming with the vertical Y-axis.

(Ox,Oy,Oz) axis are parallels to world axis. They do not depends on camera/character rotation.

1.1.2 The projection

For each side of the cubemap (front, back, bottom, top, left, right), we render a spherical depth field. The render can be small (about 32x32 pixels).The smaller it is, the fastest it will be. But if it is too small, the accuracy of the physical engine will decrease.

The projection matrices [3] for each side are :

$$A = -(z_{max} + z_{min})/(z_{max} - z_{min}) \tag{1}$$

$$B = (-2 * z_{max} * z_{min})/(z_{max} - z_{min}) \tag{2}$$

$$P_{front} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{3}$$

$$P_{back} = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -A & B \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{4}$$

$$P_{bottom} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & A & 0 & B \\ 0 & -1 & 0 & 0 \end{pmatrix} \tag{5}$$

$$P_{top} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -A & 0 & B \\ 0 & 1 & 0 & 0 \end{pmatrix} \tag{6}$$

$$P_{left} = \begin{pmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ A & 0 & 0 & B \\ -1 & 0 & 0 & 0 \end{pmatrix} \tag{7}$$

$$P_{right} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -A & 0 & 0 & B \\ 1 & 0 & 0 & 0 \end{pmatrix} \tag{8}$$

where z_{min} and z_{max} are the camera nearest and furthest planes (only points with $z_{min} < z < z_{max}$ are displayed.

1.1.3 The vertex shader

In the vertex shader, we compute clipping coordinates $gl_{Position}$ by doing :

$$gl_{Position} = P_{side}.M_{objet}.vec4(position - center, 1)$$

where M_{objet} is the movement matrix of an object of the scenery, *position* is the position of the vertex, and *center* is the center of the bounding sphere.

1.1.4 The fragment shader

In the fragment shader, we give a color for each pixel depending on the distance between the point and the center of the bounding sphere. Because each color component (R,G,B) is usually encoded on 8 bits, we clip this distance between R_{min} and R_{max}. R_{max} is the radius of the bounding sphere, and $R_{min} < R_{max}$ is the maximum penetration distance of a scenery object into the bounding sphere.

$$d = |vec3(M_{objet}.vec4(position - center, 1))| \tag{9}$$

$$red = 1. - \frac{d - R_{min}}{R_{max} - R_{min}} \tag{10}$$

$$gl_{FragColor} = vec4(red, 0., 0., 1.); \tag{11}$$

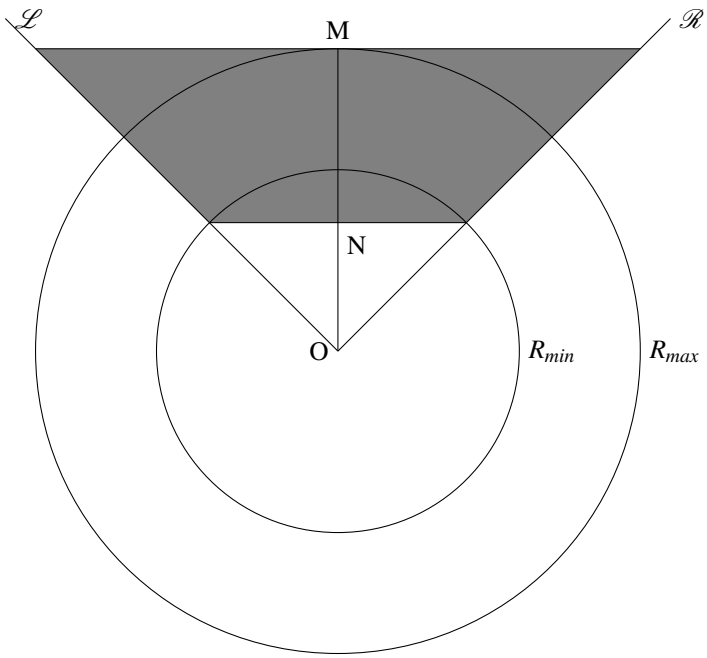

Figure 2. O is the center of the bounding sphere, $\mathscr{R}, \mathscr{L}$, represents the cubemap projection boundaries of one side. The rendering area is gray filled.

We can compute $z_{min} = ON$ and $z_{max} = OM$ from R_{min} and R_{max} by

$$z_{min} = \frac{R_{min}}{\sqrt{2}} \tag{12}$$

$$z_{max} = \sqrt{2}.R_{max} \tag{13}$$

1.2 Path correction

The cubemap provides the information about the distance between the bounding sphere and near objects with which it could be interaction. We must process this information in order to prevent the bouding sphere from colliding with objects.

1.2.1 Computing collision vectors

Firstly, we need to be able to compute θ and ϕ for each pixel of the cubemap. We compute arrays with θ and ϕ for each side of the cubemap once and for all at the loading of the application. Then we compute its cartesian coordinates x, y, z for each collision vector (point of the cubemap where the red component r is strictly positive).

$$x = r.sin(\theta) * sin(\phi) \tag{14}$$
$$y = r.cos(\phi) \tag{15}$$
$$z = r.cos(\theta) * sin(\phi) \tag{16}$$

1.2.2 Get max values

We search for the pixel with the highest red value r for each side of the cubemap. We only keep the cartesian coordinate matching with this side. For example, for the front view of the cubemap (along Z axis), we keep the z coordinate. We define :

$$x_{max} = x_{r_{max},right} \tag{17}$$
$$x_{min} = x_{r_{max},left} \tag{18}$$
$$y_{max} = y_{r_{max},top} \tag{19}$$
$$y_{min} = y_{r_{max},bottom} \tag{20}$$
$$z_{max} = z_{r_{max},back} \tag{21}$$
$$z_{min} = z_{r_{max},front} \tag{22}$$

$x_{max}, y_{max}, z_{max}$ are always positive while $x_{min}, y_{min}, z_{min}$ are negative.

For optimisation, we only need to compute the nice component of the cartesian coordinates for r_{max} points, so there are only 6 components to compute (one per cubemap side).

We will use collision with front and back cubemap view to displace the bounding sphere along Z axis, right and left cubemap view to displace it along X axis, and top and bottom view to move it along Y axis.

1.2.3 Displacement of the sphere

We define the collision response c_y along Y axis (vertical axis) differently to the collision response $c_{xz} < c_y$ along X and Z axis (horizontal axis). It is due to the gravity. We compute :

$$d_x = c_{xz} * (x_{max} + x_{min}) \tag{23}$$
$$d_y = -c_y * (y_{min} + y_{max}) \tag{24}$$
$$d_z = c_{xz} * (z_{max} + z_{min}) \tag{25}$$

Then we move the camera/the main character of $[d_x, d_y, d_z]$ vector. To allow the small crossing obstacles seamlessly (obstacles smaller than h_{max}), we add a threshold effect on d_x and d_z :

if $|d_x| < h_{max}$ then $d_x = 0$
if $|d_z| < h_{max}$ then $d_z = 0$

2. Implementation

2.1 The drawing loop

This is the pseudo-code of the rendering loop (executed for each frame). *dt* is the time period (time between 2 runs of the rendering function).

```
G=9.8 //gravity
vy=0  //vertical speed

fonction animate() {
  // displace the bounding sphere with the gravity
  vy = vy + G.dt
  dyg = vy.dt;

  // displacement is the elementary displacement
  // due to user controls (keyboard, mouse)
  move(bounding_sphere, [0, dyg, 0]+displacement)

  render the cubemap
  //compute the response of collisions
  compute [dx, dy, dz]

  if (dy<0) {
    vy=0; //the user is on the ground
  }

  // move the user
  move(user, [dx, dy, dz]+[0, dyg, 0]+displacement)

  // render the scene...
}
```

2.2 WebGL

We have implemented this algorithm with WebGL [4]. This algorithm is particularly suitable with WebGL because :

- WebGL gives only access to 2 kinds of shaders : the vertex shader and the fragment shader. We cannot use the geometry shader to compute physics,
- Of course we cannot access to specific hardware physic computation modules, like PhysX for Nvidia graphic devices,
- CPU computations are quite slow with WebGL, because they are done with Javascript.

References

[1] Jeffrey Mahovsky and Brian Wyvill. Fast Ray-Axis aligned bounding box overlap tests with plücker coordinates. *Journal of Graphics, GPU, and Game Tools*, 9(1):35+, 2004.

[2] Costas Tzafestas and Philippe Coiffet. Real-time collision detection using spherical octrees: Virtual reality application. In *IEEE Int. Work. on Robot and Human Communication*, pages 11–14, 1996.

[3] Xavier Bourry. *WebGL : Guide de développement d'applications web 3D*. Editions ENI, 2013.

[4] Khronos Group. Official webgl specifications, 2011.