Cubemap based collision detection

Xavier Bourry

Abstract
We provide an algorithm for character-scenery physics. Instead of using CPU geometry collision detection 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 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 scenery 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 pre-computed 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

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.

Figure 1. Only back and right projection planes are shown.
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:

\[
A = \frac{z_{\text{max}} + z_{\text{min}}}{(z_{\text{max}} - z_{\text{min}})} \\
B = \frac{-2z_{\text{max}}z_{\text{min}}}{(z_{\text{max}} - z_{\text{min}})} \\
\]

\[
P_{\text{front}} = \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & A & B \\
0 & 0 & -1 & 0 \\
\end{pmatrix} \\
\]

\[
P_{\text{back}} = \begin{pmatrix}
-1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & -A & B \\
0 & 0 & 1 & 0 \\
\end{pmatrix} \\
\]

\[
P_{\text{bottom}} = \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 \\
0 & A & 0 & B \\
0 & -1 & 0 & 0 \\
\end{pmatrix} \\
\]

\[
P_{\text{top}} = \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & -A & 0 & B \\
0 & 1 & 0 & 0 \\
\end{pmatrix} \\
\]

\[
P_{\text{left}} = \begin{pmatrix}
0 & 0 & -1 & 0 \\
0 & 1 & 0 & 0 \\
A & 0 & 0 & B \\
-1 & 0 & 0 & 0 \\
\end{pmatrix} \\
\]

\[
P_{\text{right}} = \begin{pmatrix}
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
-A & 0 & 0 & B \\
1 & 0 & 0 & 0 \\
\end{pmatrix} \\
\]

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

1.1.3 The vertex shader
In the vertex shader, we compute clipping coordinates \(gl\text{Position}\) by doing:

\[
gl\text{Position} = P_{\text{side}}M_{\text{object}}\text{.vec4(position - center, 1)}
\]

where \(M_{\text{object}}\) is the movement matrix of an object of the scenery, \(\text{position}\) is the position of the vertex, and \(\text{center}\) 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 \(R_{\text{min}}\) and \(R_{\text{max}}\). \(R_{\text{max}}\) is the radius of the bounding sphere, and \(R_{\text{min}} < R_{\text{max}}\) is the maximum penetration distance of a scenery object into the bounding sphere.

\[
d = |\text{vec3}(M_{\text{object}}\text{.vec4(position - center, 1)})|
\]

\[
\text{red} = 1 - \frac{d - R_{\text{min}}}{R_{\text{max}} - R_{\text{min}}}
\]

\[
gl\text{FragColor} = \text{vec4(red, 0, 0, 1)};
\]

where \(R_{\text{max}}\) and \(R_{\text{min}}\) are the center of the bounding sphere, \(\mathcal{R}\), \(\mathcal{L}\), represents the cubemap projection boundaries of one side. The rendering area is gray filled.

We can compute \(z_{\text{min}} = ON\) and \(z_{\text{max}} = OM\) from \(R_{\text{min}}\) and \(R_{\text{max}}\) by

\[
z_{\text{min}} = \frac{R_{\text{min}}}{\sqrt{2}}
\]

\[
z_{\text{max}} = \sqrt{2}R_{\text{max}}
\]

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 bounding sphere from colliding with objects.
We define the collision response vector. To allow the small crossing obstacles seamlessly, we define:

\[ x = r \sin(\theta) * \sin(\phi) \]  
\[ y = r \cos(\phi) \]  
\[ z = r \cos(\theta) * \sin(\phi) \]

1.2.1 Computing collision vectors
Firstly, we need to be able to compute \( \theta \) and \( \phi \) for each pixel of the cubemap. We compute arrays with \( \theta \) and \( \phi \) 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).

\[ x_{\text{max}} = x_{r_{\text{max}}, \text{right}} \]  
\[ x_{\text{min}} = x_{r_{\text{max}}, \text{left}} \]  
\[ y_{\text{max}} = y_{r_{\text{max}}, \text{top}} \]  
\[ y_{\text{min}} = y_{r_{\text{max}}, \text{bottom}} \]  
\[ z_{\text{max}} = z_{r_{\text{max}}, \text{back}} \]  
\[ z_{\text{min}} = z_{r_{\text{max}}, \text{front}} \]

\( x_{\text{max}}, y_{\text{max}}, z_{\text{max}} \) are always positive while \( x_{\text{min}}, y_{\text{min}}, z_{\text{min}} \) are negative.

For optimisation, we only need to compute the nice component of the cartesian coordinates for \( r_{\text{max}} \) 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.

1.2.2 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:

\[ x_{\text{max}} = x_{r_{\text{max}}, \text{right}} \]  
\[ x_{\text{min}} = x_{r_{\text{max}}, \text{left}} \]  
\[ y_{\text{max}} = y_{r_{\text{max}}, \text{top}} \]  
\[ y_{\text{min}} = y_{r_{\text{max}}, \text{bottom}} \]  
\[ z_{\text{max}} = z_{r_{\text{max}}, \text{back}} \]  
\[ z_{\text{min}} = z_{r_{\text{max}}, \text{front}} \]

\[ x_{\text{max}}, y_{\text{max}}, z_{\text{max}} \] are always positive while \( x_{\text{min}}, y_{\text{min}}, z_{\text{min}} \) are negative.

If \( x_{\text{max}} \) matches with this side. For example, for the front view of the cubemap (along Z axis), we keep the \( z \) coordinate. We define:

\[ x_{\text{max}} = x_{r_{\text{max}}, \text{right}} \]  
\[ x_{\text{min}} = x_{r_{\text{max}}, \text{left}} \]  
\[ y_{\text{max}} = y_{r_{\text{max}}, \text{top}} \]  
\[ y_{\text{min}} = y_{r_{\text{max}}, \text{bottom}} \]  
\[ z_{\text{max}} = z_{r_{\text{max}}, \text{back}} \]  
\[ z_{\text{min}} = z_{r_{\text{max}}, \text{front}} \]

1.2.3 Displacement of the sphere
We define the collision response \( c_x \) 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:

\[ d_x = c_x * (x_{\text{max}} + x_{\text{min}}) \]  
\[ d_y = -c_y * (y_{\text{min}} + y_{\text{max}}) \]  
\[ d_z = c_{xz} * (z_{\text{max}} + z_{\text{min}}) \]

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

if \( |d_x| < h_{\text{max}} \) then \( d_x = 0 \)
if \( |d_z| < h_{\text{max}} \) 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).

\[ G=9.8 \quad // \text{gravity} \]
\[ vy=0 \quad // \text{vertical speed} \]

\[ \text{fonction animate()} \{ \]
\[ // \text{displace the bounding sphere with the gravity} \]
\[ vy = vy + G \cdot dt \]
\[ dyg = vy \cdot dt ; \]
\[ // \text{displacement is the elementary displacement} \]
\[ // \text{due to user controls (keyboard, mouse)} \]
\[ \text{move(bounding_sphere, \{0, dyg, 0\}+displacement)} \]
\[ // \text{compute the response of collisions} \]
\[ \text{compute \{dx, dy, dz\}} \]
\[ \text{if } (dy<0) \{ \]
\[ \text{vy=0; //the user is on the ground} \]
\[ \} \]
\[ // \text{move the user} \]
\[ \text{move(user, \{dx, dy, dz\}+[0, dyg, 0]+displacement)} \]
\[ // \text{render the scene...} \]

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