View Frustum Culling Tutorial

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Document serial number: 000004892117

Document serial number: 000004892117

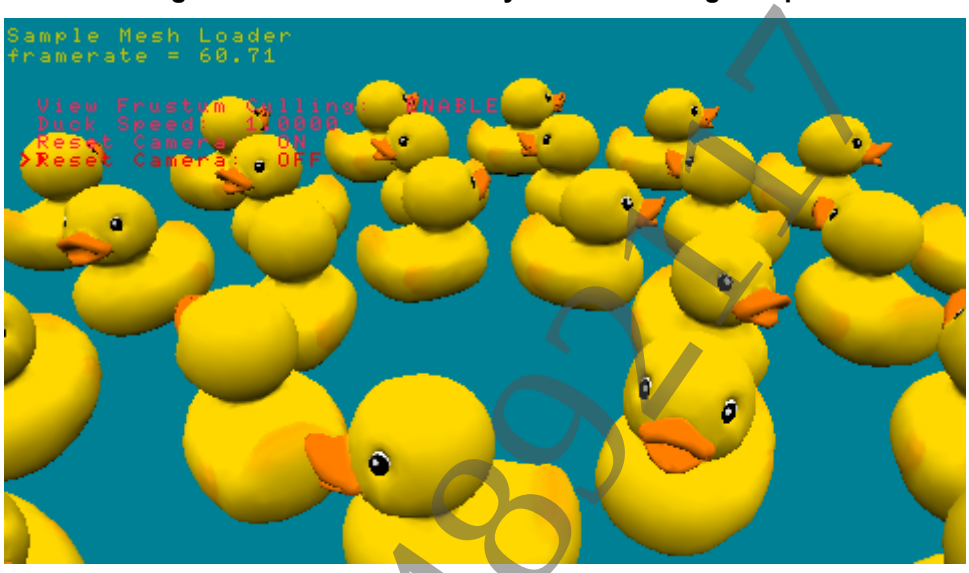
Table of Contents

1 View Frustum Culling Tutorial

Overview

This sample describes how to reduce the GPU load by using the view frustum culling technique to cull undrawn objects on the CPU side. For example, drawing performance can be improved in the scene immediately after the sample program has been executed by turning ON "View Frustum Culling" in the menu. This prevents the frame drops that occur when "View Frustum Culling" is OFF.

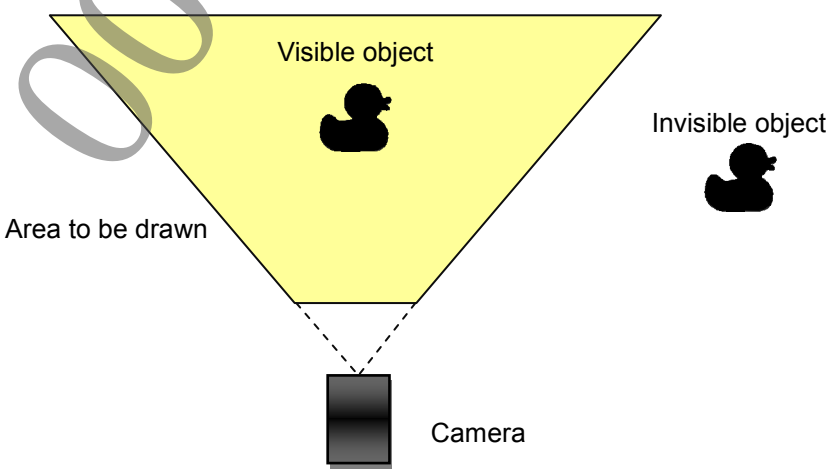
Figure 1 Scene Immediately After Executing Sample



Theory

View frustum culling is a technique used to find an object that will not be drawn in the render target within the GPU drawing process and remove it from the drawing process. This process is performed by CPU before transferring the data to the GPU, so that the GPU load can be reduced. By using view frustum culling, the GPU load can be reduced most effectively in scenes containing many objects that are not included in the drawing area.

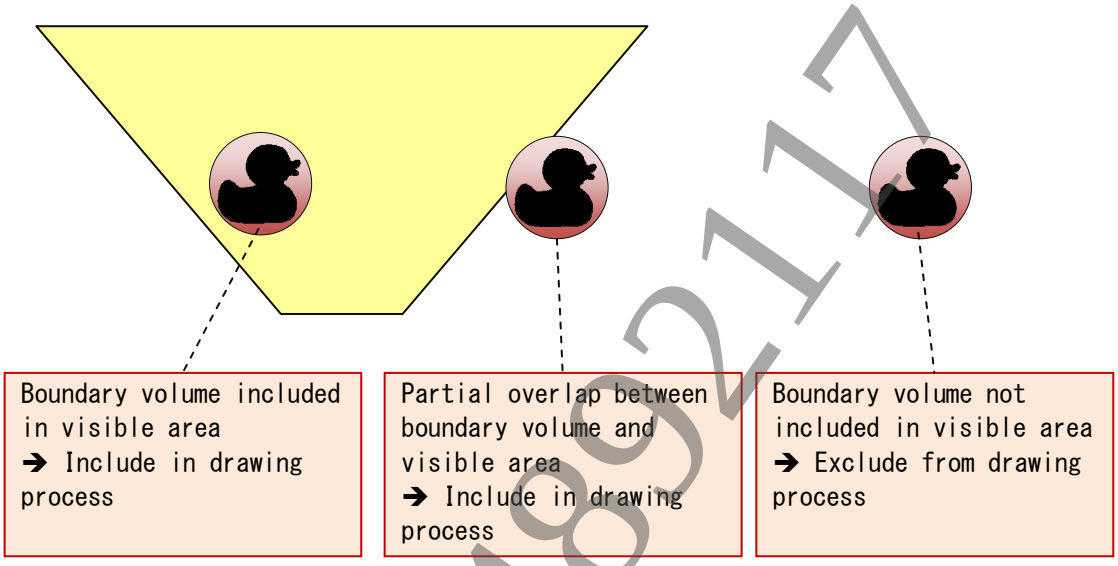
Figure 2 Overview of View Frustum Culling



If object visibility is precisely determined on the CPU side, it will increase the CPU load heavily. Therefore, on the CPU side, just a simple visibility determination is performed by utilizing the simple shaped bounding volume configured to the object. Then cull the object that definitely does not exist within the area to be drawn. This will reduce the GPU load without placing a heavy load on the CPU.

This section explains how to lighten visibility determination by the bounding volume on the CPU side. First, configure the bounding volume as a sphere or another simple shape so that it will completely contain the object. Visibility determination is performed by detecting the intersection between the bounding volume and the camera's field of vision. If the bounding volume and field of vision overlap, it will be necessary to determine which part of the object will be drawn using detailed visibility determination on the GPU. Configure the entire object so that it is included in the GPU drawing process.

Figure 3 Intersection Detection by the CPU



2 Implementation

This section explains how implement view frustum culling is implemented in the sample.

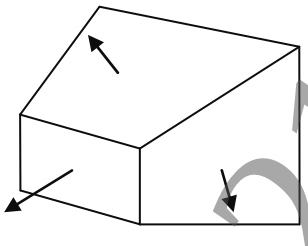
Data Model

Visible Area

The visible area of the camera matches the target drawing area of the GPU and will be shown using frustum. This is called view frustum.

Frustum is shown on 6 planes. The interior and exterior of each plane is configured using normal vectors.

Figure 4 Frustum



Bounding Volume

This sample adopts a bounding sphere for the bounding volume in which the object will be configured so that the intersection with a plane can rapidly be detected.

Detecting an Intersection Between Visible Area and Bounding Volume

To detect an intersection between the visible area and bounding volume, an intersection must first be detected between the 6 planes that compose the view frustum and the bounding volume. If the bounding volume does not intersect with the interior of any plane, the bounding volume is determined to be outside of the visible area.

```
class BoundingSphere : public sce::Geometry::Aos::Sphere;
class ViewFrustum : public sce::Geometry::Aos::Frustum;

/* test the bounding sphere intersects to with bounding sphere. */
static bool isIntersect(const ViewFrustum &frustum,
const BoundingSphere &boundingSphere)
{
  Vectormath::boolInVec ret =
      sce::Geometry::Aos::isIntersecting(frustum, boundingSphere);
  return ret.getAsBool();
}
```

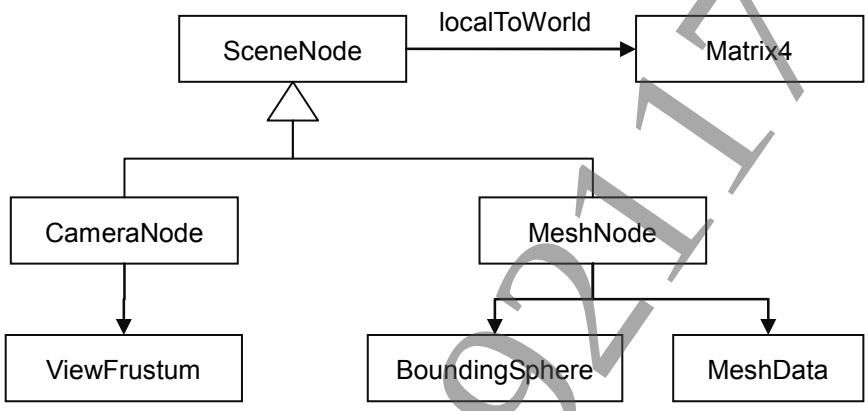
Data Structure

The 3D scene is configured by the camera and the object.

The camera and the object are shown as nodes within the 3D space. Each node contains a matrix that converts the local coordinate system centered on that node to the world coordinate system that is shared within the scene.

In addition to this matrix, camera nodes contain the view frustum in the local coordinate system of a camera node, and mesh nodes that can be drawn such as objects contain the mesh data and bounding sphere data in the local coordinate system.

Figure 5 Class Diagram of Main Data



Procedure

The following procedure is performed for each frame.

(1) Update Location

With `SampleFrustumCullingApplication::update()`, update the localToWorld matrix of the mesh node and camera.

```
m_innerDuckNode[i]->setLocalToWorld(
            Matrix4::translation(pos)
            * Matrix4::rotationY(- angle - SCE_MATH_PI/2.0f));
```

(2) Visibility Determination

The `preRender()` method is called within `SampleFrustumCullingApplication::render()` for each node. Calculate the position of the boarder sphere within the camera's local coordinate system to perform intersection detection using the following formula:
Inverse matrix of the camera's transformation matrix × Transformation matrix of the mesh node.

```
virtual void preRender(scene::Scene* scene)
{
   Vectormath::Aos::Matrix4 vm = scene->getCamera()->getViewMatrix();
   geometory::BoundingSphere tbs =
         geometory::BoundingSphere::transform(vm * localToWorld,
                                              boundingSphere);
   isVisible = scene->getCamera()->isVisible(tbs);
}
```

(3) Draw

Only draw objects that intersect with the view frustum.

```
void render(Scene* scene, SceGxmContext *pContext)
{
    if(! isVisible){
       return;
    }
    /* Draw object */
    …
}
```