Tutorial for Special 3D Render Techniques

- Level of Detail (LOD)
- Morphing
- Planar Shadows
- Baked Shadows
- Single Pass Effects in Candera > Examples for 3D Effect Shaders
- Multi Pass Effects in Candera > Examples for Local 3D Effect Shaders
- Multi Pass Effects in Candera > Examples for Global 3D Effect Shaders
- Transform Feedback

Level of Detail (LOD)

Description

This chapter briefly describes Level Of Detail techniques supported by Candera.

See also:

- Level of Detail in SceneComposer User Manual

Example Solution:

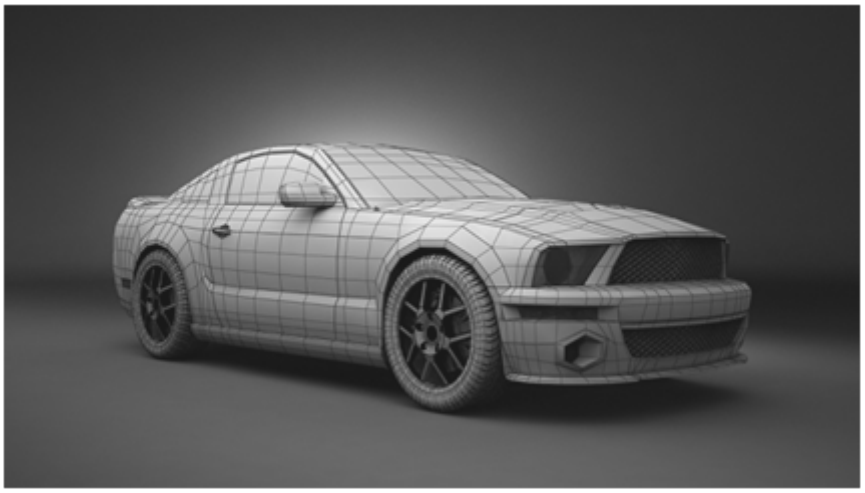
- LODSolution from folder *cgi_studio_player/content/Tutorials/08_SpecialRenderTechniques/Special3DRenderTechniques*

Introduction

Level of Detail

```
Level of detail (LOD) is a discipline of interactive computer graphics attempts
to bridge complexity and performance by regulating the amount of detail used
to represent the virtual world.
```

(Level of Detail for 3D Graphics, David Luebke, Morgan Kaufmann Publisher, 2003)



Motivation

Reduce render complexity in order to increase runtime performance with minimal or without visual penalty

Detail Simplifications

- Geometric: Meshes, Imposters, Shadow LOD, ...
- Non-geometric: Shader, Lighting, Textures (MipMapping), Material, ...

Level of Detail - Geometry Reduction Types

Discrete LOD

- Creates multiple versions of an object with different level of details. Often refered to as isotropic or view-independent LOD, as detail reduction is applied uniformly accross the object.

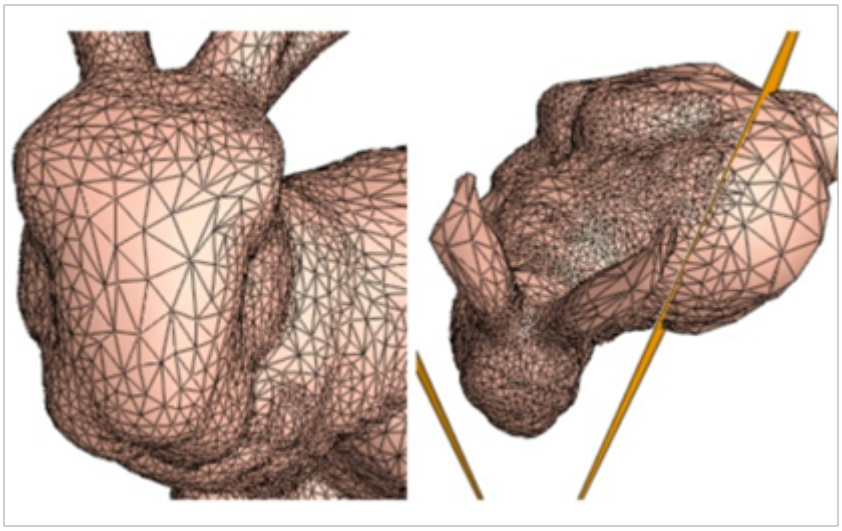
- Widely used in real time environments as LOD levels are generated during an offline preprocess. Supported by Candera.

Continuous or progressive LOD

- Simplification system creates a data structure encoding a continuous spectrum of detail. The desired LOD is extracted from this structure at run-time. Supports LOD streaming and interruptible loading.

View-dependent LOD

- Extends continuous LOD by using view-dependent simplification criteria to dynamically select the most appropriate LOD for current view. Thus, view-dependent view is anisotropic.



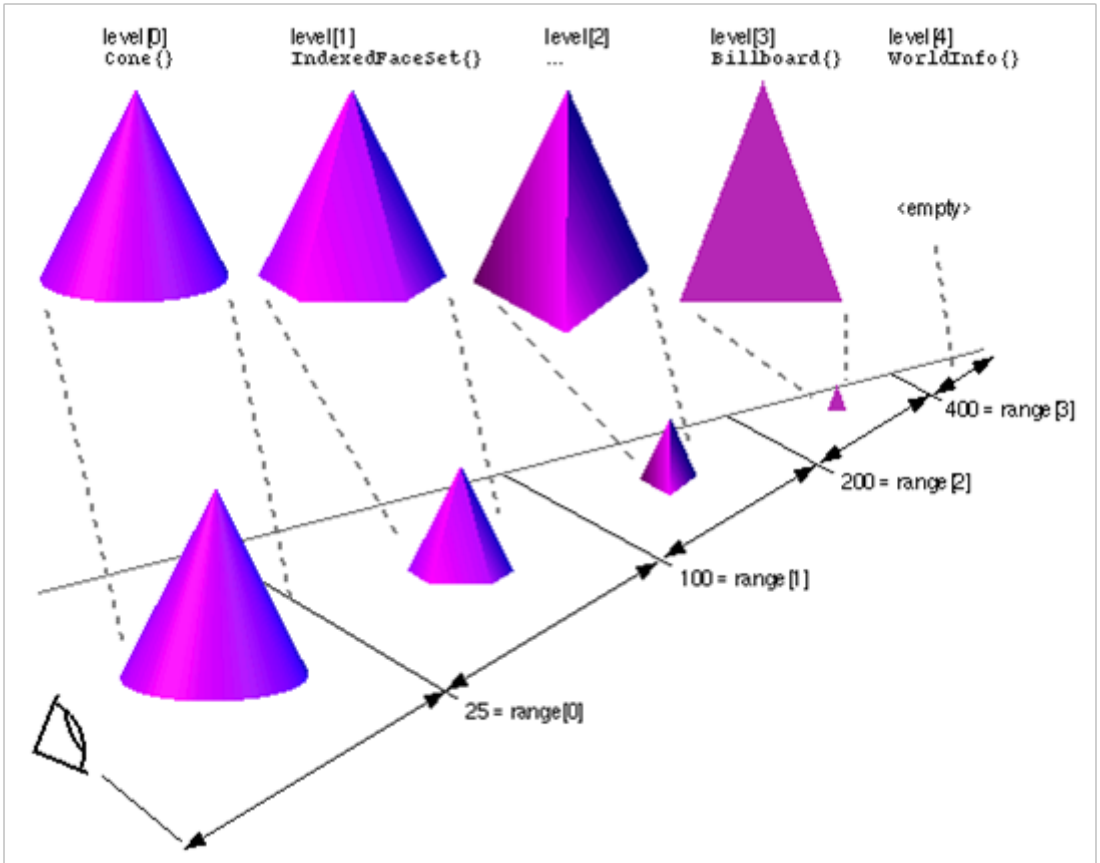
Level of Detail - Chain

- A LodNode accomodates a node for each level of detail.
- Candera Nodes are e.g: Mesh, Billboard, Light, Group, etc.

Typical LOD Chain

The following figure depicts a typical LOD chain:

- Level 0 .. 2: Meshes.
- Level 3: Billboard.
- Level 4: null-Node



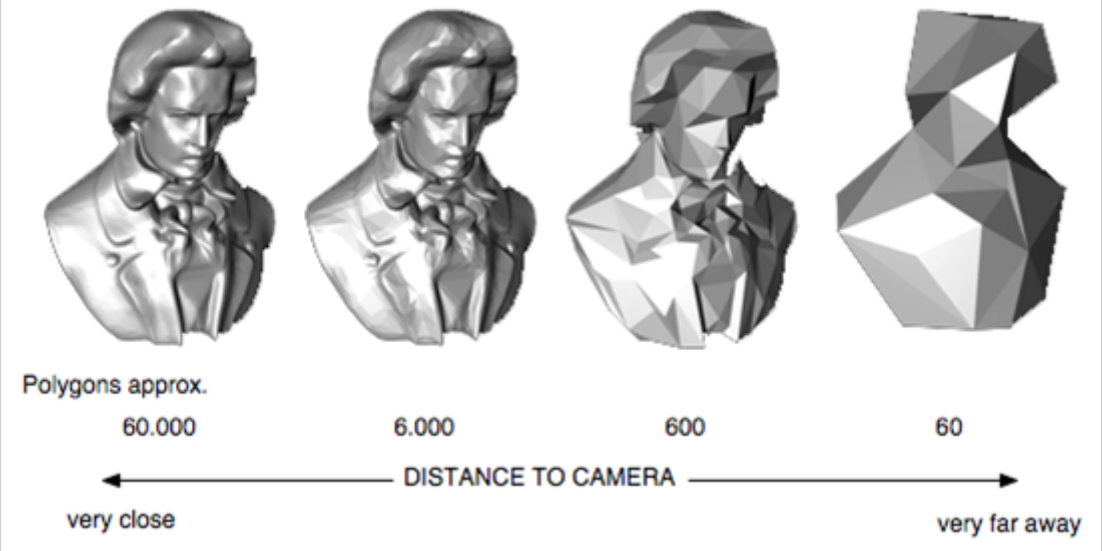
Candera - Discrete LOD Concept

Candera::LodNode, LoD Level

The class Candera::LodNode implements discrete LOD technique. Any Candera class derived from class Candera::Node is accepted as LOD representation.

Each LOD level in Candera::LodNode defines:

- LOD level index,
- Node,
- Lower bound of its visibility range,
- Upper bound of its visibility range.



LOD Render Strategy

The LOD render strategy defines the transition between adjacent LOD levels.

- Candera::DiscreteLodRenderStrategy: Stepwise transition.
- Candera::BlendLodRenderStrategy: alpha-blended transition.

LOD Criterion

The LOD criterion delivers a source value to map onto the LOD nodes visibility boundaries.

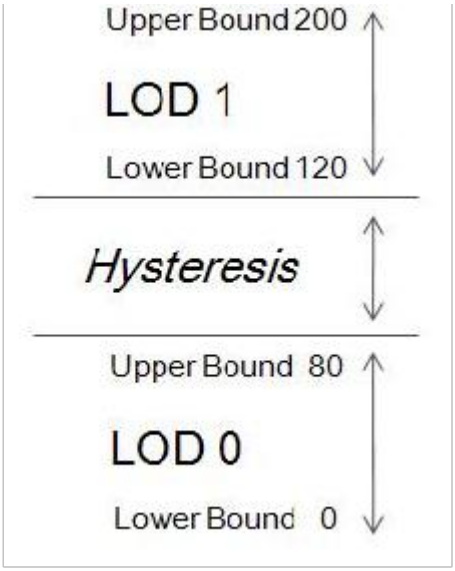
- Candera::DistanceToCameraLodCriterion: distance from LOD node to camera.

- Candera::GenericValueLodCriterion: Application defined value.

Candera - LOD Render Strategy

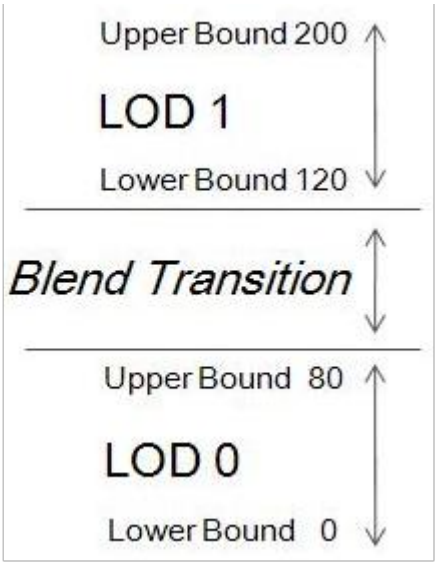
Discrete LOD Render Strategy

- Stepwise transition between LOD levels
- Non-adjacent boundaries between contiguous LOD levels define a Hysteresis.



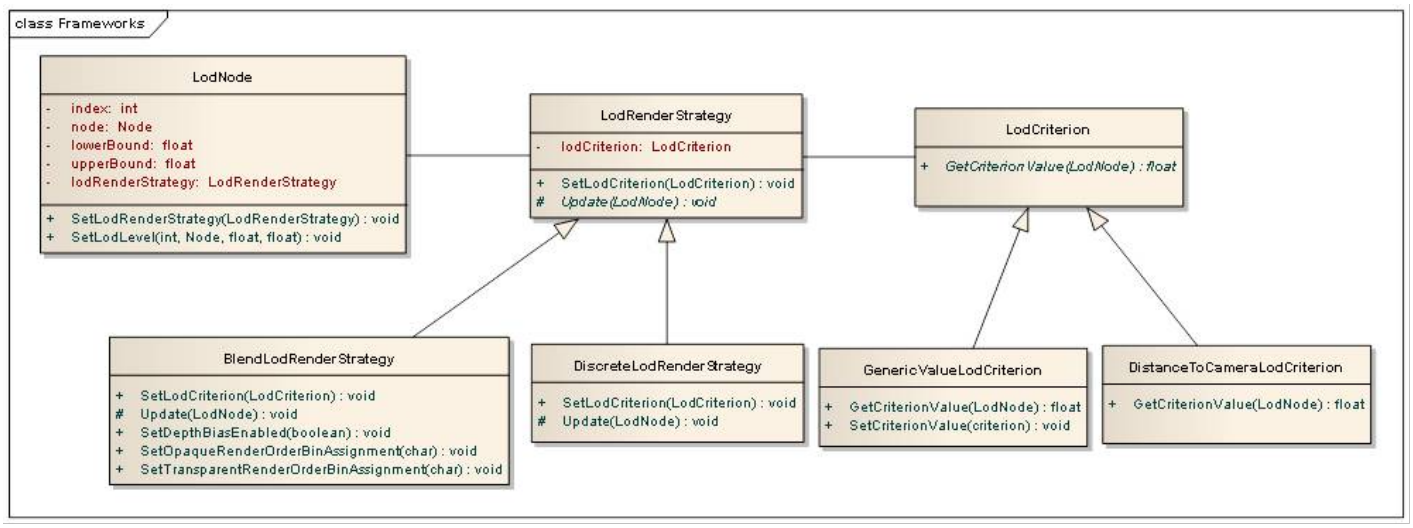
Blend Transition LOD Render Strategy

- Alpha blended transitions between LOD levels
- Non-adjacent boundaries between contiguous LOD levels define transition phase.
- During blend transition one alternating LOD node is always opaque to avoid 'see-through' effects.



Candera - LOD Architecture Overview

- A Candera::LodNode has 0..1 associated LodRenderStrategy objects.
- A Candera::LodRenderStrategy has 0..1 associated LodCriterion objects.
- According to use case LodRenderStrategy and LodCriterion objects are interchangeable arbitrarily.



Example - LOD in SceneComposer

Example - Level of Detail in SceneComposer

This chapter shows a typical LOD use case and explains how the transition from two different LODs can be realized with CGI Studio SceneComposer.

To experiment with LOD transition, the following can be used:

- **LODSolution** from folder *cgi_studio_player/content/Tutorials/08_SpecialRenderTechniques/0801_Special3DRenderTechniques*

Use Case

The example solution is based on following use case:

- The tire is shown from a certain distance in sufficient resolution.
- The camera zooms in to the tire (animation).
- As the camera captures the tire from a close distance, the tire model shall be exchanged with a higher resolution model.
- The transition between the low- and the high-resolution tire model shall appear smooth.

You can experiment with following node properties:

- **(Lower and Upper) Boundary Values**: Modify boundary values of low and high node.
- **Criterion**: Set criterion 0:DistanceToCamera 1:Generic.
- **Strategy**: Set render strategy 0:Blend 1:Discrete.

See also:

- Level of Detail in SceneComposer User Manual

Example - Blended LOD

Blended LOD

- Initialize LOD node with 2 LOD objects.
- LOD is automatically changed by distance to camera criterion.
- Desired Behavior:
  - LOD 0 is exclusively visible at camera distance 0 - 40 (in the LodNode set Lower Bound 0, and Upper Bound 40).
  - LOD 1 is exclusively visible at camera distance 60 - 100 (in the LodNode set Lower Bound 60, and Upper Bound 60).
  - Blend Transition between LOD 0 and LOD 1 is defined at camera distance 40 - 60 (in the LodNode select RenderStrategy 0:Blend).

Change CameraDistance or start CameraPosition animation to see the smooth transition.

In application code the LodNode is set up as follows:

Initializing the LodNode

```
    // init LodNode bounds
    static_cast<void>(lodNode->SetLodLevel(0, lodNode->GetLodLevel(0).node, m_lowBoundHigh, m_up
    static_cast<void>(lodNode->SetLodLevel(1, lodNode->GetLodLevel(1).node, m_lowBoundLow, m_upE
    // end init
```

Define Criterion and RenderStrategy

```
        // Define DistanceToCameraLodCriterion and BlendLodRenderStrategy

        static DistanceToCameraLodCriterion cameraLodCriterion; // LOD is automatically selected
        static BlendLodRenderStrategy blendLodStrategy; // LOD is automatically blended between

        // End Define DistanceToCameraLodCriterion and BlendLodRenderStrategy
```

Link pieces together

```
        // Init BlendLodRenderStrategy and set to LodNode

        blendLodStrategy.SetLodCriterion(&cameraLodCriterion);
        lodNode->SetLodRenderStrategy(&blendLodStrategy);

       // End Init BlendLodRenderStrategy and set to LodNode
```

Example - Step-wise LOD

Step-wise LOD

- Initialize LOD node with 2 LOD objects.
- LOD criterion value can be set by application directly. (in the LodNode set Criterion to 1:Generic).
- Desired Behavior:
  - LOD 0 is exclusively visible at camera distance 0 - 40 (in the LodNode set Lower Bound 0, and Upper Bound 40).
  - LOD 1 is exclusively visible at camera distance 60 - 100 (in the LodNode set Lower Bound 60, and Upper Bound 60).
  - Hysterese is defined at 40 - 60. Thus, criterion values within that range do not lead to LOD level swaps. (in the LodNode select RenderStrategy 1:DiscreteLodRenderStrategy).

Change CriterionValue property appropriate to see the transition.

In application code the LodNode is set up as follows:

Initializing the LodNode

```
    // init LodNode bounds
    static_cast<void>(lodNode->SetLodLevel(0, lodNode->GetLodLevel(0).node, m_lowBoundHigh, m_up
    static_cast<void>(lodNode->SetLodLevel(1, lodNode->GetLodLevel(1).node, m_lowBoundLow, m_upB
    // end init
```

Define Criterion and RenderStrategy

```
        // Define GenericLodCriterion and DiscreteLodRenderStrategy

        static GenericValueLodCriterion genericLodCriterion; // Application defined criterion va
        static DiscreteLodRenderStrategy discreteLodStrategy; // Stepwise LOD switch.
```

```
        // End Define GenericLodCriterion and DiscreteLodRenderStrategy
```

Link pieces together

```
        // Init DiscreteLodRenderStrategy and set to LodNode

        genericLodCriterion.SetCriterionValue(m_criterionValue); // Set application defined crit
        discreteLodStrategy.SetLodCriterion(&genericLodCriterion);
        lodNode->SetLodRenderStrategy(&discreteLodStrategy);

        // End Init DiscreteLodRenderStrategy and set to LodNode
```

Example - Set LOD directly

Set LOD directly

- Create LOD node with 2 LOD objects.
- LOD level index is set by application directly.
- Hint: Direct LOD level initialization is helpful, if first criterion value falls into hysteresis range, and no LOD is set active.

```
// Create LodNode object with 2 empty LOD levels.
m_lodNode = LodNode::Create(2);
// Set LOD 0 active and deactivate LOD 1. No RenderCriterion is attached.
m_lodNode->GetLodLevel(0).node->SetRenderingEnabled(true);
m_lodNode->GetLodLevel(1).node->SetRenderingEnabled(false);
```

Morphing

Description

This chapter briefly describes the morphing technique supported by Candera.

See also:

- Morphing Mesh in SceneComposer User Manual

Example Solutions

Refer to following solutions in folder *cgi_studio_player/content/Tutorials/08_SpecialRenderTechniques/0801_Special3DRenderTechniques*

- MorphSolution
- MultiMorphing_MultiMorphShader
- MultiMorphing_AnimationGroup

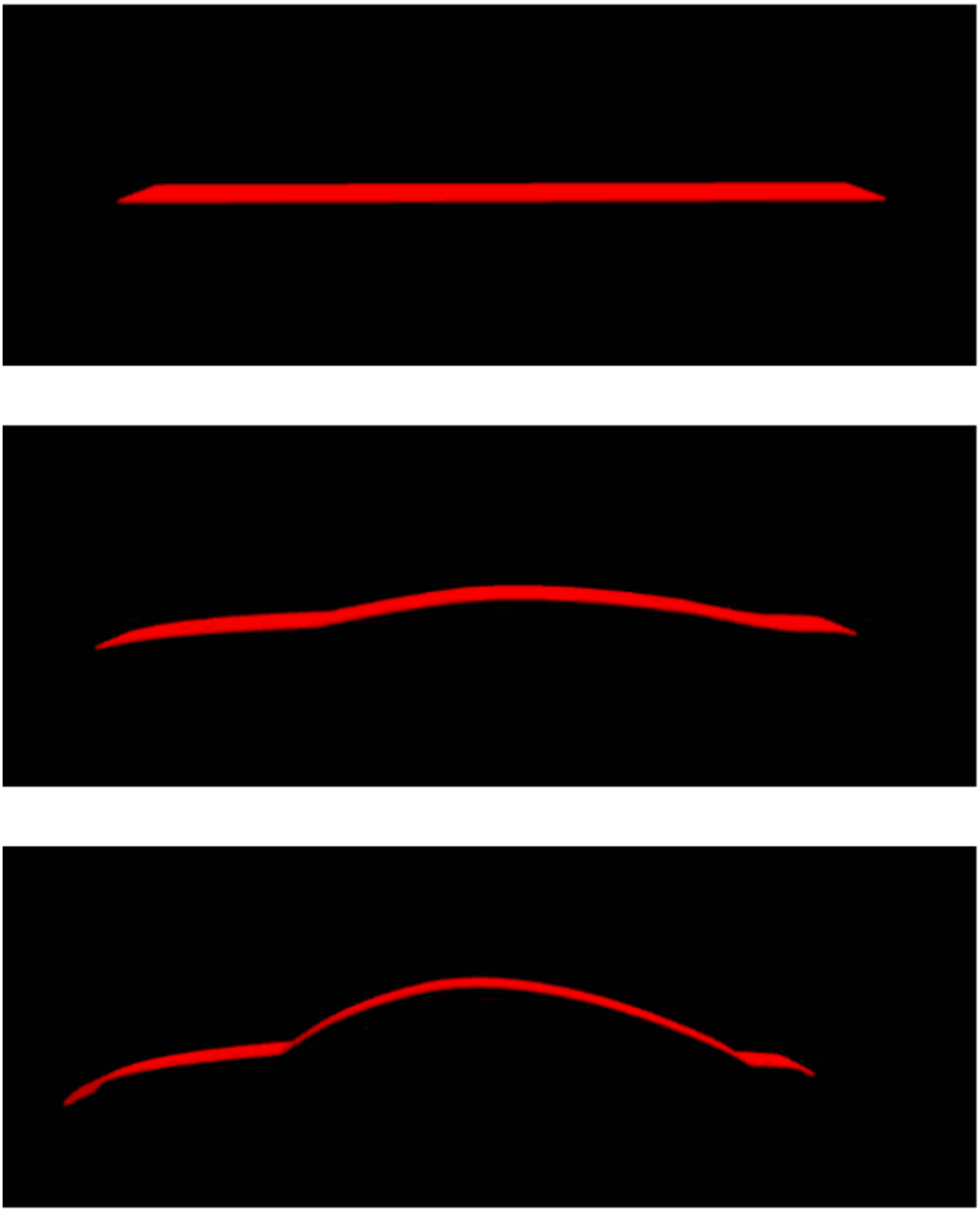
Introduction

Morphing

Morphing means transforming the 3D geometry of an object smoothly from one shape into another.

Morphing itself (smooth interpolation between vertices) is done in a shader (not on the CPU).

Morping example



What you need for Morphing

Suitable content: "Morph targets" are different geometric states of the same object. The input vertices must correspond one-to-one between morph targets. This means, to morph between two shapes, you need **two VertexBuffer** with exactly the **same number of vertices**.

An instance of MorphingMesh: This is a Mesh that stores

- the spliced vertex buffer created from the vertex buffers to be morphed (see MorphingMesh::CreateSplicedVertexBuffer)

- weight values for each of the shapes, which are activated as shader uniforms at render time (see MorphingMesh::SetMorphWeight)
- The appearance of the MorphingMesh must define a suitable morphing shader, which matches the number of shapes to be morphed (see **RefTransLight1Morph** vertex shader for morphing two shapes)

Dynamics: Morphing is about change, so an application will have to set some morphing parameters (often depending on time). Use Animation::AllMorphWeightsPropertySetter to animation all the weight values of the MorphingMesh.

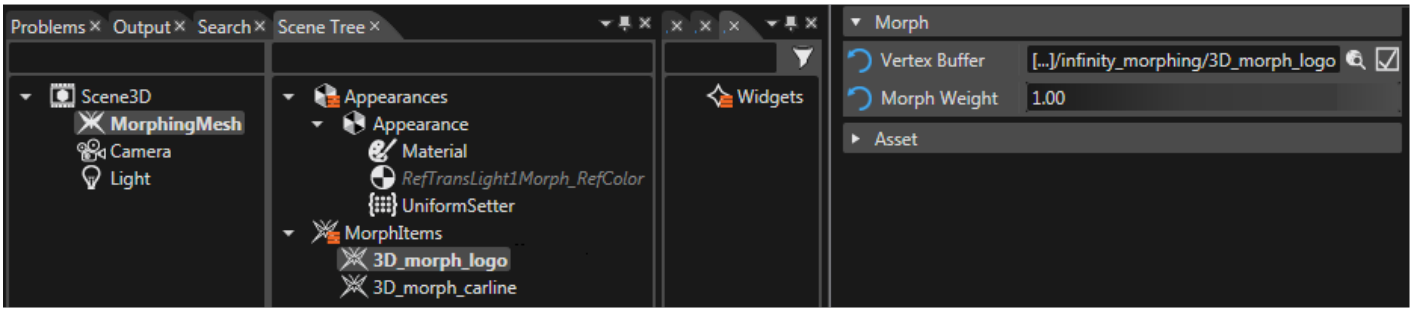
Morphing Examples

Morphing between 2 Morph Items

In below example solution, the reference shader **RefTransLight1Morph_RefColor** is used to morph between two morph items, by default having weight value 1,00 on the logo.

- *MorphSolution* from folder *cgi_studio_player/content/Tutorials/08_SpecialRenderTechniques/0801_Special3DRenderTechniques*

MorphingMesh setup for two morphing shapes



The animation part of this solution animates both weight values and also adds a rotation animation.

Morphing between Multiple Shapes using Shader

Since a MorphingMesh is capable of handling up to 8 different vertex buffers, a special morphing shader must be used for more than 2 shapes. A morphing shader handling 5 shapes is used in the example below:

- *MultiMorphing_MultiMorphShader* from folder *cgi_studio_player/content/Tutorials/08_SpecialRenderTechniques/0801_Special3DRenderTechniques*

Shader code for morphing 5 shapes

TransLight1Morph5.vert

```
449
450   /*---------------------------------- MAIN ----------
451   void main(void)
452   {
453       vec3 position =
454           a_Position4.xyz * u_MorphWeight[4]
455         + a_Position3.xyz * u_MorphWeight[3]
456         + a_Position2.xyz * u_MorphWeight[2]
457         + a_Position1.xyz * u_MorphWeight[1]
458         + a_Position.xyz  * u_MorphWeight[0];
459       vec3 norm =
460         normalize(
461           a_Normal4.xyz * u_MorphWeight[4]
462         + a_Normal3.xyz * u_MorphWeight[3]
463         + a_Normal2.xyz * u_MorphWeight[2]
464         + a_Normal1.xyz * u_MorphWeight[1]
465         + a_Normal.xyz  * u_MorphWeight[0]);
466       v_TexCoord =
467           a_TextureCoordinate4.st * u_MorphWeight[4]
468         + a_TextureCoordinate3.st * u_MorphWeight[3]
469         + a_TextureCoordinate2.st * u_MorphWeight[2]
470         + a_TextureCoordinate1.st * u_MorphWeight[1]
471         + a_TextureCoordinate.st  * u_MorphWeight[0];
```

Morphing between Multiple Shapes using AnimationGroup

As an alternative to using a multi morph shader, multi-step morphing can also be realized by using an AnimationGroup, chaining separate morphing animations for each morphing shape pair. This approach is useful in case a specific target shader compiler has some restrictions regarding the maximum number of shader parameters.

- *MultiMorphing_AnimationGroup* from folder *cgi_studio_player/content/Tutorials/08_SpecialRenderTechniques/0801_Special3DRenderTechniques*

Planar Shadows

Description

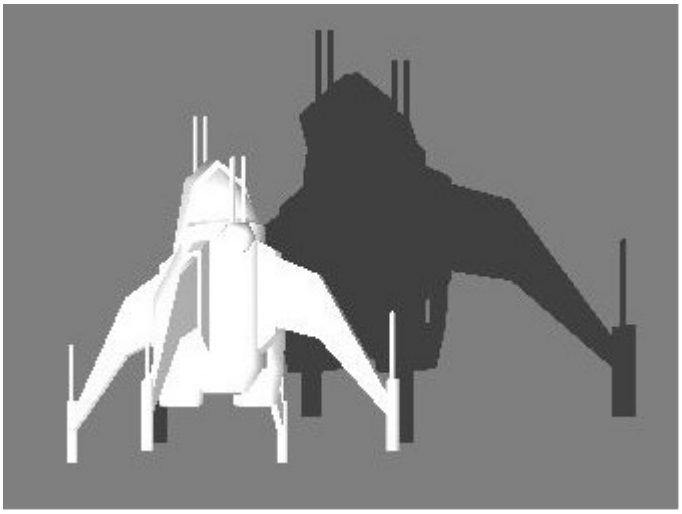
This chapter describes the planar shadow technique supported by Candera.

Introduction

Planar Shadows

Planar shadows are used to algorithmically simulate flat shadows on planar surfaces.

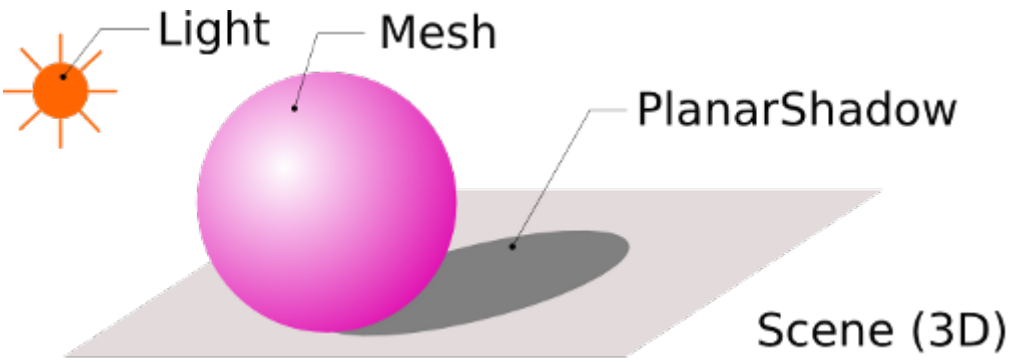
The shadow effect is obtained using a Candera::PlanarShadow object which is drawn using its world transformation matrix multiplied with a projection matrix, that projects its vertices onto a predefined plane with respect to the position of an associated light source.



Workflow

Inventory

The figure below shows the basic setup of a scene needed to generate the planar shadow effect.



Light

Light should be of type point, spot or directional.

```
m_light = Light::Create();
m_light->SetName("Light1");
m_light->SetType(Light::Point);
m_light->SetDiffuse(Color(1.0f, 1.0f, 1.0f));
m_light->SetSpecular(Color(0.0f, 0.0f, 0.0f));
m_light->SetRenderingEnabled(true);
```

Shadow Caster

The following objects can be used as shadow caster: Candera::LineList, Candera::Billboard and Candera::Mesh.

Planar Shadow

The Candera::PlanarShadow is the object which actually draws the shadow onto a specified plane. By default, the parent node's vertex buffer is used. Consequently the PlanarShadow node should be attached as child of the shadow caster.

```
m_shadow = PlanarShadow::Create();
m_shadow->GetAppearance()->SetShader(m_shaderShadow); // RefTrans_RefUniDiffuseMat
m_shadow->SetName("Shadow");
m_shadow->SetPlane(Plane(Vector3(0.0f, 0.0f, 1.0f), 75.0f));
m_shadow->SetLight(m_light);
m_current->AddChild(m_shadow); // attach planar shadow to the shadow caster
```

It's also possible to explicitly provide a dedicated vertex buffer which will be rendered as shadow, see the snippet below:

```
                m_shadow->SetAutoVertexBufferEnabled(false);
                m_shadow->GetVertexBuffer()->Unload();
                m_shadow->SetVertexBuffer(m_meshes[1]->GetVertexBuffer());
                m_shadow->GetVertexBuffer()->Upload();
```

Candera::PlanarShadow supports also an optional alignment of the shadow plane to a given node (alignmentNode). If this node it set, the shadow plane's distance and direction are transformed locally according to alignmentNode's transformation matrix. If node is not set, the shadow plane's parameters are not transformed at all. See snipped below to set alignmentNode:

```
m_shadow->SetAlignmentNode(m_meshes[3]);
```

Overlapping Shadows

To avoid double blending of semi transparent shadows stencil buffers can be used. The PlanarShadow's appearance already contains a pre-configured RenderMode with stencil buffers configured to only draw on fragments with a stencil value other than this PlanarShadows unique Id. After drawing, the value gets replaced to the current stencil reference value, such that the stencil test doesn't pass any more on future fragments to write.

> To get this mechanism running ensure that the camera always clears the stencil buffer to 0 every frame.

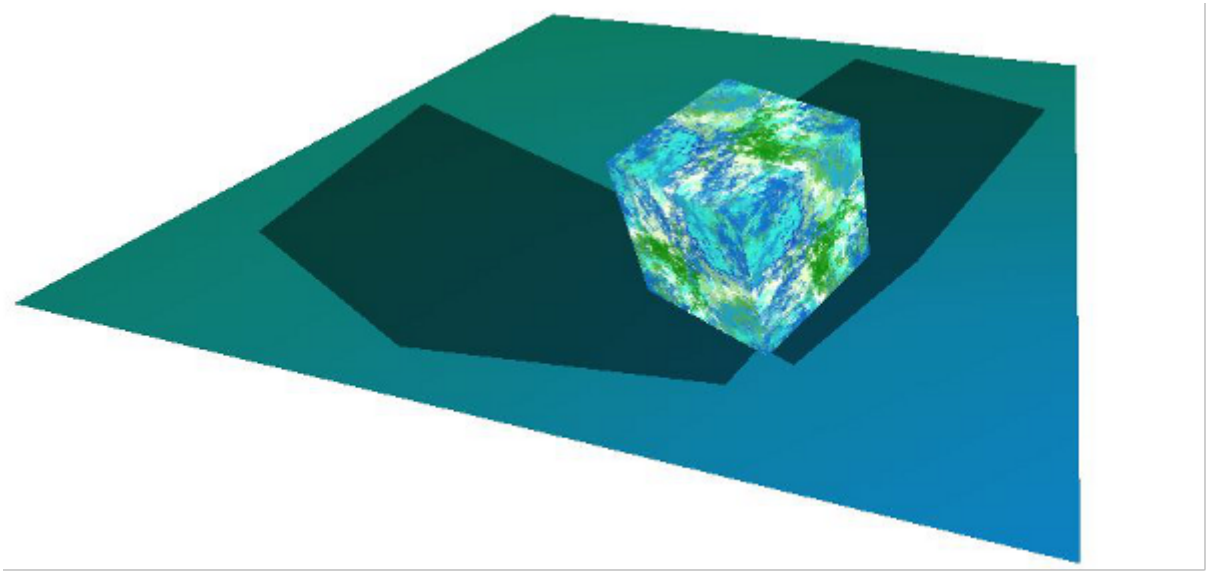
> The GraphicDeviceUnit has to be initialized accordingly to support stencil buffers.

Constraints

- Only flat planar surfaces are considered as areas to draw the shadow on.
- Every object that intends to cast a shadow, must have a dedicated shadow child node for each light source, as well as for each plane acting as shadow receiver.
- If vertex buffer transformations shall be considered (e.g. morphing meshes or displacement mapping), a dedicated shader has to be implemented for the shadow object.

Example

The example shows two planar shadows casts by a cube.



Planar Shadow Example in SceneComposer

To experiment with the Planar Shadow in SceneComposer use solution **PlanarShadow** from folder *cgi_studio_player/content/Tutorials/08_SpecialRenderTechniques/Special3DRenderTechniques*.

Baked Shadows

Description

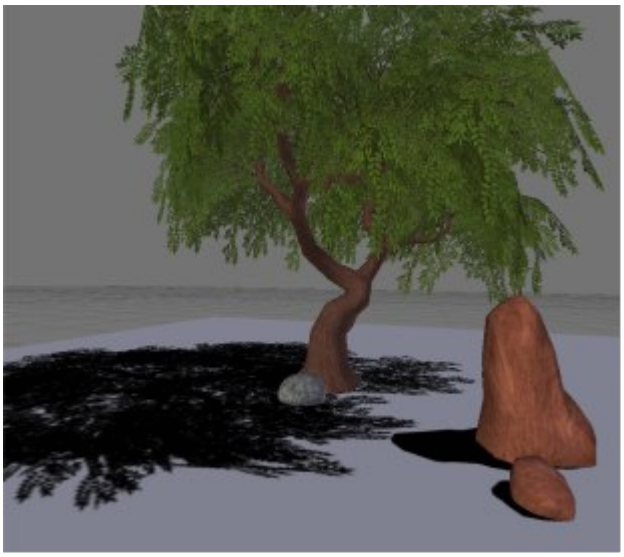
This chapter describes the baked shadow technique supported by Candera.

Introduction

Baked shadows

Baking shadows is a method of generating shadows which can be used for scenes where the lights and the objects which cast shadows are not moved with respect to each other.

Because the shadows are pre-computed and integrated into the textures the polygon count is decreased and the time needed for rendering the scene is significantly reduced. As a drawback, if the textures resulted after the baking are large, the memory can be intensive so the rendering becomes slow. Also, for complex scenes, this way of generating shadows might be too laborious to be considered as an option.[1]

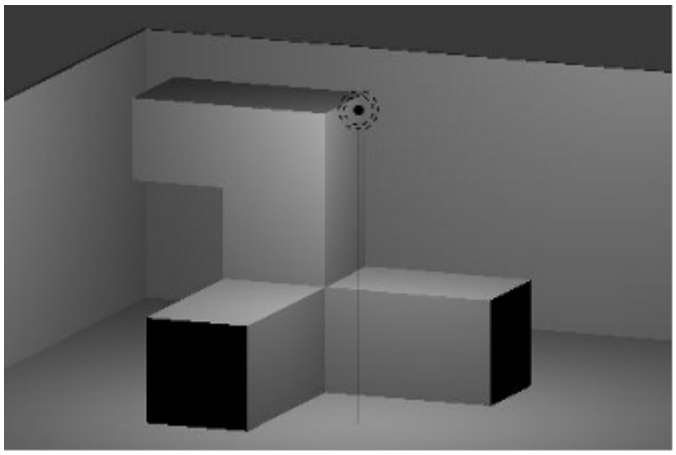


Workflow

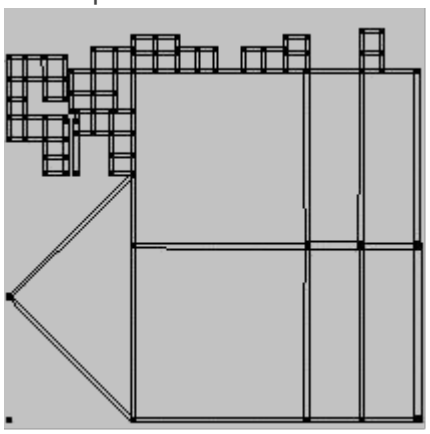
Create Models and Textures

The following steps should be performed in any 3D modeling tool that is able to bake shadows (Blender, Maya, 3D Studio Max ...):

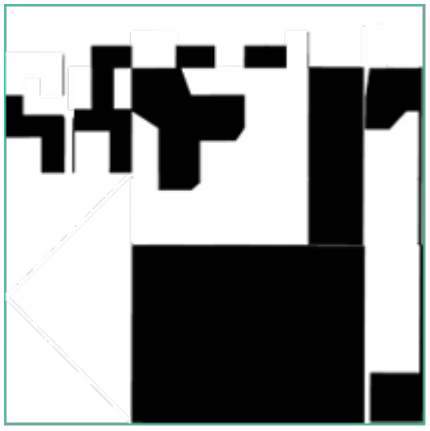
- create a 3D scene that contains some models and at least one source of light



- join the models which casts shadows with the models on which the shadow is projected on
- unwrap the 3D models



- create a new image that will be used then as the texture containing the shadows
- set the image as texture for the models
- render the scene and bake the shadows into the texture



- export the texture in any image format supported by Candera and also export the 3D models in FBX format
- If needed, refine the baked image with an image editor

Asset Library Generation

Use SceneComposer to generate the asset library:

- import the FBX and the image texture in SceneComposer
- from the Imports panel add the light and the node in a 3D scene
- set the imported image as texture for the node
- export the asset library

Example

The example shows a 3D node textured with an image containing baked shadows.

References

The pictures and the models from this tutorial where obtained from the following sources:

[1] http://apricot.blender.org/wp-content/uploads/2008/06/sbake1.jpg

Single Pass Effects in Candera > Examples for 3D Effect Shaders

Description

This chapter describes special 3D effect shaders techniques supported by Candera.

Bump Mapping

Description

This chapter briefly describes bump mapping techniques supported by Candera.

Introduction

Bump mapping is a technique in computer graphics for simulating bumps and wrinkles on the surface of an object.

This is achieved by perturbing the surface normals of the object and using the perturbed normal during lighting calculations. The result is an apparently bumpy surface rather than a smooth surface although the surface of the underlying object is not actually changed.

Workflow

Inventory

In order to generate the bump mapping effect on a surface you need:

- A mesh
- A specific shader
- A shader parameter setter
- A normal map texture
- A color map texture
- A material

Mesh

Any mesh can be used for this purpose but the mesh vertex buffer has to contain also the tangents and the binormals vertex attributes. This information can be added to the vertex buffer in the 3D content creation tool (3DS Max, Blender, etc.), before exporting the FBX, or, at run time, using the Candera::Math3D::CreateTangentsAndBinormalsFromVertexBuffer function as shown below:

```
    mesh->SetVertexBuffer(Math3D::CreateTangentsAndBinormalsFromVertexBuffer(mesh->GetVertexBuf
```

Shader

Use the shader from table below:

Vertex shader	RefTransLight1BumpMap
Fragment shader	RefLight1BumpMap

Set the appearance shader for the mesh:

```
    mesh->GetAppearance()->SetShader(bumpmapShader);
```

Shader Parameter Setter

The uniform setter will have to be set as follows:

```
    shaderParamSetter->SetModelMatrix4Enabled(true);
    shaderParamSetter->SetNormalModelMatrix3Enabled(true);
    shaderParamSetter->SetModelViewProjectionMatrix4Enabled(true);
    shaderParamSetter->SetLightActivationEnabled(true);
    shaderParamSetter->SetMaterialActivationEnabled(true);
    shaderParamSetter->SetTextureActivationEnabled(true);
    shaderParamSetter->SetLightsCoordinateSpace(Light::World);
    shaderParamSetter->SetCameraPositionEnabled(true);
    mesh->GetAppearance()->SetShaderParamSetter(shaderParamSetter);
```

Color Texture

The color texture applies an image to the surface of a mesh. This texture should be set on the mesh with texture unit 0.

Add the color texture to the appearance:

```
    mesh->GetAppearance()->SetTexture(colorTexture, 0);
```

Normal Map Texture

The normal map images store the direction of normals directly in the RGB values of an image. These normals are used for lighting calculation instead of the vertices normals. Normal maps are generated using special graphics software usually from a higher resolution geometry than the geometry you're applying the map to.[1]

Add the normal map texture to the appearance:

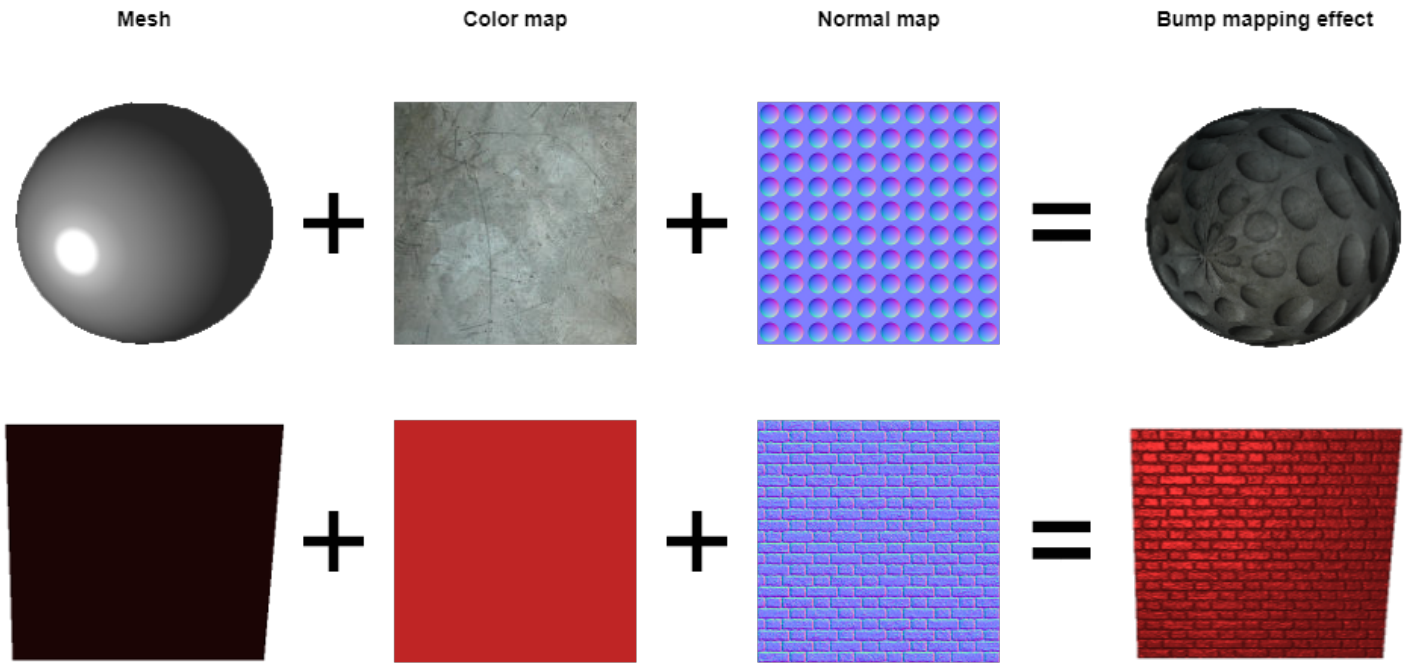
```
    mesh->GetAppearance()->SetTexture(normalMapTexture, 1);
```

Material Configuration

```
        mesh->GetAppearance()->GetMaterial()->SetAmbient(Color(0.3f, 0.3f, 0.3f, 1.0f));
        mesh->GetAppearance()->GetMaterial()->SetEmissive(Color(0.0f, 0.0f, 0.0f, 1.0f));
        mesh->GetAppearance()->GetMaterial()->SetDiffuse(Color(1.0f, 1.0f, 1.0f, 1.0f));
        mesh->GetAppearance()->GetMaterial()->SetSpecular(Color(1.0f, 1.0f, 1.0f, 1.0f));
        mesh->GetAppearance()->GetMaterial()->SetSpecularPower(40.0f);
```

Examples

Here is exemplified the bump map effect applied on two meshes:



Bump Mapping in SceneComposer

To experiment with Bump mapping in Scene Compose use the solution **ShaderExamples** from the folder *cgi_studio_player/content/Tutorials/04_ShaderUsage*.

References

The pictures from this tutorial where obtained from the following sources:

[1] http://planetpixelemporium.com/tutorialpages/normal3.html

Procedural Textures

Description

This chapter briefly describes the procedural wood technique supported by Candera.

Introduction

Procedural Wood

Procedural wood is texturing technique which generates an on-the-fly texture using an algorithm that creates a wood realistic 3D representation. The algorithm will generate concentric rings colored alternatively with two colors which, if chosen rightly, will show a wood-like appearance. The effect can be controlled by setting the uniforms:

- u_CustomWoodCenter - rings center position
- u_CustomWoodColor1 - even rings color
- u_CustomWoodColor2 - odd rings color

Workflow

Inventory

In order to generate the wood effect on a mesh all you need to do is to configure the material, an appropriate shader and a shader parameter setter for its appearance.

Shader

The table below presents the vertex and fragment shader which generates the wood effect:

Vertex shader	RefTransLight1ProceduralWood
Fragment shader	RefProceduralWood

Set the appearance shader for the mesh:

```
        mesh->GetAppearance()->SetShader(woodShader);
```

Shader Parameter Setter

The uniforms setter will have to be configured as follows:

```
        woodShaderParamSetter->SetModelViewProjectionMatrix4Enabled(true);
        woodShaderParamSetter->SetLightActivationEnabled(true);
        woodShaderParamSetter->SetMaterialActivationEnabled(true);
        woodShaderParamSetter->SetTextureActivationEnabled(true);

        mesh->GetAppearance()->SetShaderParamSetter(woodShaderParamSetter);
```

Prepare the uniforms data:

```
        Float woodCenter[] = { 0.0f, 0.0f, 0.2f};
        Float woodColor1[] = { 0.81f, 0.63f, 0.41f, 1.0f };
        Float woodColor2[] = { 0.56f, 0.27f, 0.12f, 1.0f };
        Float woodMultiplier = 100.0f;
```

Set the uniforms using the shader parameter setter:

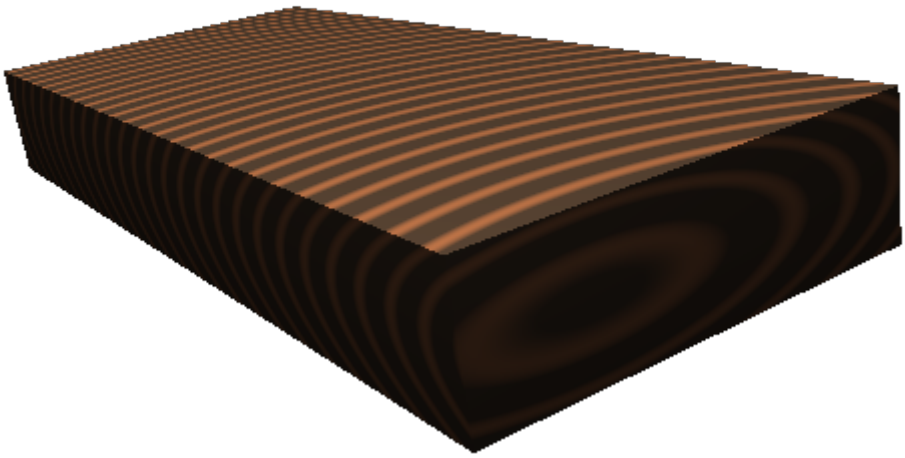
```
        woodShaderParamSetter->SetUniform("u_CustomWoodCenter", Shader::FloatVec3, woodCenter);
        woodShaderParamSetter->SetUniform("u_CustomWoodColor1", Shader::FloatVec4, woodColor1);
        woodShaderParamSetter->SetUniform("u_CustomWoodColor2", Shader::FloatVec4, woodColor2);
        woodShaderParamSetter->SetUniform("u_CustomWoodMultiplier", Shader::Float, &woodMultipli
```

Material Configuration

```
        mesh->GetAppearance()->GetMaterial()->SetAmbient(Color(0.0f, 0.0f, 0.0f, 1.0f));
        mesh->GetAppearance()->GetMaterial()->SetEmissive(Color(0.0f, 0.0f, 0.0f, 1.0f));
        mesh->GetAppearance()->GetMaterial()->SetDiffuse(Color(1.0f, 1.0f, 1.0f, 1.0f));
        mesh->GetAppearance()->GetMaterial()->SetSpecular(Color(1.0f, 1.0f, 1.0f, 1.0f));
        mesh->GetAppearance()->GetMaterial()->SetSpecularPower(40.0f);
```

Examples

Here is exemplified the wood effect applied on two meshes:



Wood Effect in SceneComposer

To experiment with the procedural wood technique in SceneComposer use solution **ShaderExamples** from folder *cgi_studio_player/content/Tutorials/04_ShaderUsage*.

References

[1] Image taken from http://blenderartists.org/forum/showthread.php?246113-A-fine-procedural-wood-material-for-Cycles

Environment Mapping

Description

This chapter describes the environment mapping techniques supported by Candera.

Environment mapping is an image based technique to achieve an appearance of reflecting object surface. A texture is used to show the environment as a reflection.

Sphere Map

Description

This chapter describes the sphere mapping techniques supported by Candera.

Sphere mapping was the first environment mapping technique where the reflective environment is mapped onto a single texture as it would be reflected by a mirror ball. Sphere mapping is suitable on concave surfaces while else Cube mapping achieves better results.

Example Usage of Sphere Mapping

Sphere Map Texture Used

SphereMap in SceneComposer

To experiment with the sphere mapping technique in SceneComposer use solution **ShaderExamples** from folder *cgi_studio_player/content/Tutorials/04_ShaderUsage*.

Cube Map

Description

This chapter describes the cube mapping techniques supported by Candera.

Cube mapping is a rendering technique whereby six bitmaps are mapped to the sides of a cube, to produce environment mapping effects, such as reflections, or a skybox background. Unlike the old sphere mapping technique, cube map images do not need to be distorted to match the current view-angle, and is therefore a more efficient and flexible method to achieve reflections. However, in OpenGL ES 2.0, any texture filtering is applied separately to the six individual images, which may cause rendering artifacts along the image borders. This issue was corrected in OpenGL ES 3.0, which provides seamless cubemap filtering. There is nothing you need to do to enable seamless cubemap filtering. All linear filter kernels will automatically use it, when targeting OpenGL ES 3.0.

A special application for cube mapping is the Skybox.

CubeMap in SceneComposer

To experiment with CubeMap in SceneComposer, use solution **ShaderExamples** from folder *cgi_studio_player/content/Tutorials/04_ShaderUsage*.

SkyBox

Description

This chapter describes the skybox technique supported by Candera.

SkyBox was implemented to support cubic panorama backgrounds ("SkyBoxes"). A Skybox displays the background environment of a scene, by using a world-space axis-aligned cube, with six images assigned to the faces of the cube. This cube is centered on the active camera's current position, which creates the illusion that the background image is infinitely far away. The skybox is rendered before any other nodes, with depth writing and testing turned off, to ensure that it appears behind all other nodes in the scene.

In OpenGL ES 2.0, linear filtering is applied separately to the six images, which may cause rendering artifacts (seams) along the cube edges. In OpenGL ES 3.0, this minor defect was corrected, by applying filtering across image borders, to create seamless cubemaps. No action is required to enable seamless cubemaps under OpenGL ES 3.0. However, this feature is not available in OpenGL ES 2.0 or its extensions.

SkyBox in SceneComposer

To experiment with SkyBox in SceneComposer use solution **ShaderExamples** from folder *cgi_studio_player/content/Tutorials/04_ShaderUsage*.

Reflection & Refraction

Description

This chapter describes the reflection and refraction techniques supported by Candera.

Candera uses new shaders to achieve reflection and refraction appearances:

- RefTransCubeMapReflection.vert
- RefTransCubeMapRefraction.vert
- RefTransCubeMapReflectionRefraction.vert
- RefCubeMapTex.frag
- RefCubeMapTex2.frag

These shaders demonstrate how environment mapping reflections, refraction and the combination of both can be realized using CubeMap textures. The combined reflection and refraction shading results in a glass effect especially when combined with a skybox using the same cubemap textures

Reflection and Refraction in SceneComposer

To experiment with Reflection and Refraction in SceneComposer use solution **ShaderExamples** from folder *cgi_studio_player/content/Tutorials/04_ShaderUsage*.

Anisotropic Lighting

Description

This chapter briefly describes the anisotropic lighting technique supported by Candera.

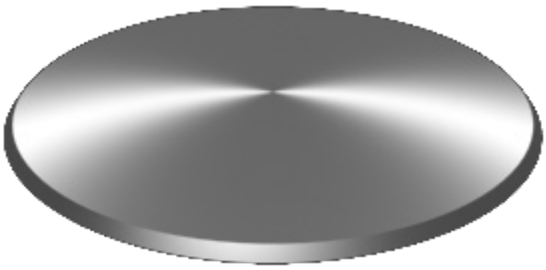
Introduction

Anisotropic Lighting

The Candera implementation of the anisotropic lighting model can be used to generate the effect of a surface having grooves or fine directional grain like, for instance, brushed metal, shiny side of a CD, vinyl records etc.

The effect can be configured by setting the uniforms:

- u_AlphaU - distribution of specular light component in U direction
- u_AlphaV - distribution of specular light component in V direction



Workflow

Inventory

In order to generate the brushed metal effect on a mesh you need to configure the material, a texture, an appropriate shader and a shader parameters setter for its appearance.

Shader

The table below presents the vertex and fragment shader which generates the brushed metal effect:

Vertex shader	RefTransAnisotropicLight1
Fragment shader	RefAnisotropicLight1SpecularTex

Set the appearance shader for the mesh:

```
        mesh->GetAppearance()->SetShader(anisotropicLightingShader);
```

Shader Parameter Setter

The uniforms setter will have to be configured as follows:

```
        m_anisotropicLightSetter->SetModelMatrix4Enabled(true);
        m_anisotropicLightSetter->SetNormalModelMatrix3Enabled(true);
        m_anisotropicLightSetter->SetModelViewProjectionMatrix4Enabled(true);
        m_anisotropicLightSetter->SetCameraPositionEnabled(true);
        m_anisotropicLightSetter->SetLightActivationEnabled(true);
        m_anisotropicLightSetter->SetMaterialActivationEnabled(true);
        m_anisotropicLightSetter->SetTextureActivationEnabled(true);
        m_anisotropicLightSetter->SetLightsCoordinateSpace(Light::World);

        mesh->GetAppearance()->SetShaderParamSetter(m_anisotropicLightSetter);
```

Prepare the uniforms data:

```
        Float m_alphaU = 0.28f;
        Float m_alphaV = 0.23f;
```

You can generate other effects like plastic laminate, glossy grey paper, rolled aluminium etc. by simply changing the values for m_alphaU, m_alphaV and the diffuse and specular colors for the material.

Set the uniforms using the shader parameters setter:

```
        m_anisotropicLightSetter->SetUniform("u_AlphaU", Shader::Float, &m_alphaU);
        m_anisotropicLightSetter->SetUniform("u_AlphaV", Shader::Float, &m_alphaV);
```

Material Configuration

```
        mesh->GetAppearance()->GetMaterial()->SetAmbient(Color(0.0f, 0.0f, 0.0f, 1.0f));
        mesh->GetAppearance()->GetMaterial()->SetEmissive(Color(0.0f, 0.0f, 0.0f, 1.0f));
        mesh->GetAppearance()->GetMaterial()->SetDiffuse(Color(0.4f, 0.4f, 0.4f, 0.4f));
        mesh->GetAppearance()->GetMaterial()->SetSpecular(Color(1.0f, 1.0f, 1.0f, 1.0f));
        mesh->GetAppearance()->GetMaterial()->SetSpecularPower(40.0f);
```

Texture

Set the appearance texture:

```
        mesh->GetAppearance()->SetTexture(texture, 0);
```

Example

The picture below shows the final result after setting the mesh appearance as described above:

Multi Pass Effects in Candera > Examples for Local 3D Effect Shaders

Description

This chapter describes special 3D effect shaders techniques supported by Candera.

Fur

Description

This chapter describes how to generate the fur effect using the multipass appearance technique supported by Candera.

Introduction

The fur effect is produce by rendering the node multiple times with different appearances which are blended together. Each appearance will use the same combination of noise and color map textures but with different and specific render states. For each appearance the following uniforms will have to be set specifically: [1]

- u_ColorScale - fur color scale.
- u_Scale - scale factor to manipulate vertices along normals
- u_PassIndex - render pass index

Workflow

Appearances

Because the generation of the fur effect implies that the node has to be rendered multiple times, a special appearance needs to be used: Candera::MultiPassAppearance. The particularity of this appearance is that it keeps a pointer to the next appearance, defining so a sequence of render passes.

> A nice fur effect is obtained with at least 10 render passes.

```
for (int i = 0; i<= 15; i++) {
    m_furAppearances[i] = MultiPassAppearance::Create();
    // Configure the appearances
    // ...
    // End of appearances configuration
    if (i > 0) {
        m_furAppearances[i-1]->SetNextPass(m_furAppearances[i]);
    }
}
mesh->SetAppearance(m_furAppearances[0]);
```

Shader

The table below presents the vertex and fragment shader needed to generate the fur effect:

Vertex shader	RefTransFur
Fragment shader	RefUniColorTexFur

Set this shader for all the appearances:

```
m_furAppearances[i]->SetShader(m_furShader);
```

> Lighting and material has no influence on the fur effect.

Shader Parameters Setter

Configure the uniforms setter as follows:

```
        m_furSetter[i] = GenericShaderParamSetter::Create();
        m_furSetter[i]->SetModelViewProjectionMatrix4Enabled(true);
        m_furSetter[i]->SetLightActivationEnabled(true);
        m_furSetter[i]->SetMaterialActivationEnabled(true);
        m_furSetter[i]->SetTextureActivationEnabled(true);
```

Prepare the uniforms data:

```
        m_furColorScale[0] = 0.2196f;
        m_furColorScale[1] = 0.2202f;
        m_furColorScale[2] = 0.2202f;
        m_furColorScale[3] = 1.0f;

        if ( i == 0 ) {
            m_furScale = 1.0f;
        }
        else {
            m_furScale = 0.01f;
        }
        m_passIndex[i] = ((Float) i);
```

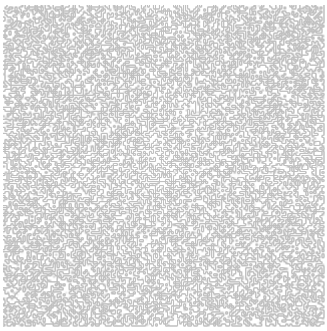
Set the uniforms using the shader parameter setter:

```
        m_furSetter[i]->SetUniform("u_ColorScale",Shader::FloatVec4,m_furColorScale);
        m_furSetter[i]->SetUniform("u_Scale",Shader::Float,&m_furScale);
        m_furSetter[i]->SetUniform("u_PassIndex",Shader::Float,&m_passIndex[i]);
```

Material Configuration

```
        if ( i == 0) {
            m_furAppearances[i]->SetMaterial(Material::Create());
        }
        else {
            m_furAppearances[i]->SetMaterial(m_furAppearances[0]->GetMaterial());
        }
```

Noise texture



Set the fur appearance noise texture:

```
    m_furAppearances[i]->SetTexture(m_furNoise, 0);
```

Color map texture

The color map texture is meant to provide a color for the fur.

Set the fur appearance color map texture:

```
    m_furAppearances[i]->SetTexture(m_furColor, 1);
```

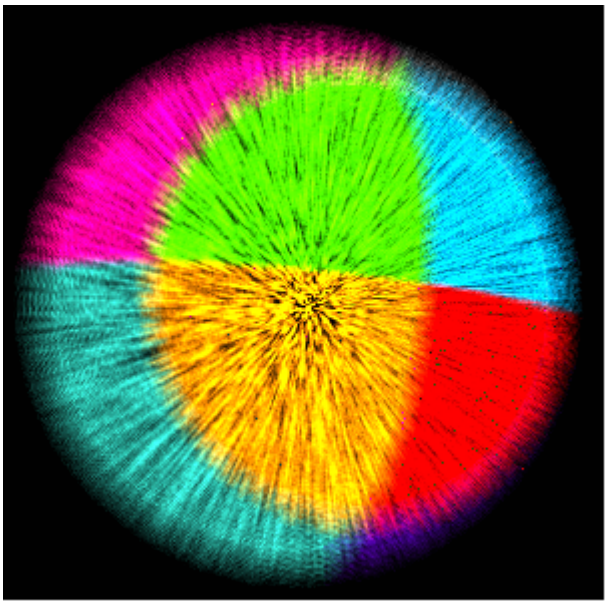
Render Mode

Set and configure a Render Mode for all transparency layers (appearances 1 to 15).

```
    if (i > 0) {
        m_furAppearances[i]->GetRenderMode()->SetBlendingEnabled(true);
        m_furAppearances[i]->GetRenderMode()->SetBlendMode(RenderMode::SourceAlpha, RenderMode::
    }
```

Example

Here is exemplified the fur effect applied on sphere:



Fur Effect in SceneComposer

To experiment with the fur effect in SceneComposer use solution **ShaderExamples** from folder *cgi_studio_player/content/Tutorials/04_ShaderUsage*.

References

[1] Image taken from http://www.xbdev.net/directx3dx/specialX/Fur/index.php

---

Gooch Effect

Description

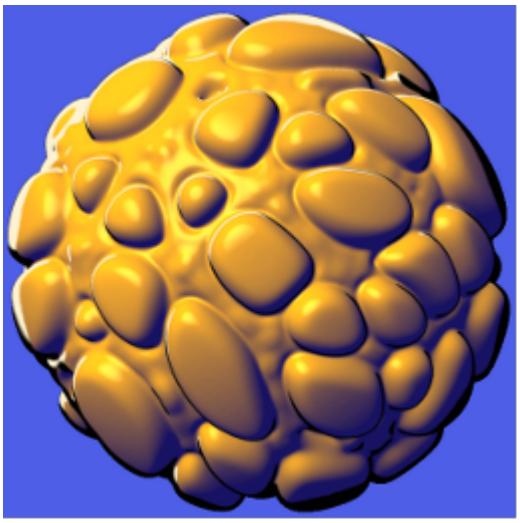
This chapter describes how to generate the Gooch effect using the multipass appearance technique supported by Candera.

Introduction

Gooch Effect

The Gooch effect is a non photorealistic technique which simulates the appearance of technical illustration.

The effect is generated in two render passes. In the first pass the object is drawn in warm - cool colors: all surfaces that are facing toward the light source are drawn in warm colors (red, yellow etc), all surfaces away from the light source are drawn in cool colors (blue, violet etc.). The second render pass draws the silhouette of the object with a fixed color (e.g. black). [1]



Workflow

Inventory

In order to generate the Gooch effect on a node you need:

- Two MultiPassAppearances
- Specific shader for each appearance
- Specific shader parameters setter for each appearance
- Material
- Render Modes

Appearances

The Gooch effect is generated in two render passes so we need two MultiPassAppearances:

```
    m_goochAppearance1 = MultiPassAppearance::Create();
    m_goochAppearance2 = MultiPassAppearance::Create();
    m_goochAppearance1->SetNextPass(m_goochAppearance2);

    m_mesh->SetAppearance(m_goochAppearance1);
```

Shader

Each appearance need a specific shader, see table below:

Appearance	Vertex shader	Fragment shader
m_goochAppearance1	RefTransLight1Gooch	RefGoochShading
m_goochAppearance2	RefTransScale	RefUniColor

Set the shader for each appearance:

```
    m_goochAppearance1->SetShader(m_goochPass1Shader);
    m_goochAppearance2->SetShader(m_goochPass2Shader);
```

Shader Parameter Setter

The uniform setter will have to be set as follows:

```
    m_goochUniformSetter1 = GenericShaderParamSetter::Create();
    m_goochUniformSetter1->SetMaterialActivationEnabled(true);
    m_goochUniformSetter1->SetLightActivationEnabled(true);
    m_goochUniformSetter1->SetLightsCoordinateSpace(Light::World);
    m_goochUniformSetter1->SetModelViewProjectionMatrix4Enabled(true);
    m_goochUniformSetter1->SetModelMatrix4Enabled(true);
    m_goochUniformSetter1->SetNormalModelMatrix3Enabled(true);
    m_goochUniformSetter1->SetCameraPositionEnabled(true);
    m_goochAppearance1->SetShaderParamSetter(m_goochUniformSetter1);

    m_goochUniformSetter2 = GenericShaderParamSetter::Create();
    m_goochUniformSetter2->SetModelViewProjectionMatrix4Enabled(true);
    m_goochAppearance2->SetShaderParamSetter(m_goochUniformSetter2);
```

Prepare uniform data:

```
    m_coolDiffuseWeight= 0.25f;
    m_warmDiffuseWeight = 0.5f;
    m_goochScale = 0.1f;

    m_coolColor[0] = 0.0f;
    m_coolColor[1] = 0.0f;
    m_coolColor[2] = 0.55f;
    m_coolColor[3] = 1.0f;

    m_warmColor[0] = 0.3f;
    m_warmColor[1] = 0.3f;
    m_warmColor[2] = 0.0f;
    m_warmColor[3] = 1.0f;

    m_silhouetteColor[0] = 0.0f;
    m_silhouetteColor[1] = 0.0f;
    m_silhouetteColor[2] = 0.0f;
    m_silhouetteColor[3] = 1.0f;
```

Set the uniforms:

```
    m_goochUniformSetter1->SetUniform("u_CoolDiffuseWeight", Shader::Float, &m_coolDiffuseWeight
    m_goochUniformSetter1->SetUniform("u_WarmDiffuseWeight", Shader::Float, &m_warmDiffuseWeight
    m_goochUniformSetter1->SetUniform("u_CoolColor", Shader::FloatVec4, m_coolColor);
    m_goochUniformSetter1->SetUniform("u_WarmColor", Shader::FloatVec4, m_warmColor);

    m_goochUniformSetter2->SetUniform("u_Scale", Shader::Float, &m_goochScale);
    m_goochUniformSetter2->SetUniform("u_Color", Shader::FloatVec4, m_silhouetteColor);
```

Material Configuration

```
    m_goochAppearance1->SetMaterial(Material::Create());
    m_goochAppearance1->GetMaterial()->SetDiffuse(Color(0.75f, 0.75f, 0.75f, 1.0f));
    m_goochAppearance1->GetMaterial()->SetAmbient(Color(0.0f, 0.0f, 0.0f, 0.0f));
    m_goochAppearance1->GetMaterial()->SetSpecular(Color(1.0f, 1.0f, 1.0f, 1.0f));
    m_goochAppearance1->GetMaterial()->SetSpecularPower(20.0f);


    m_goochAppearance2->SetMaterial(Material::Create());
```

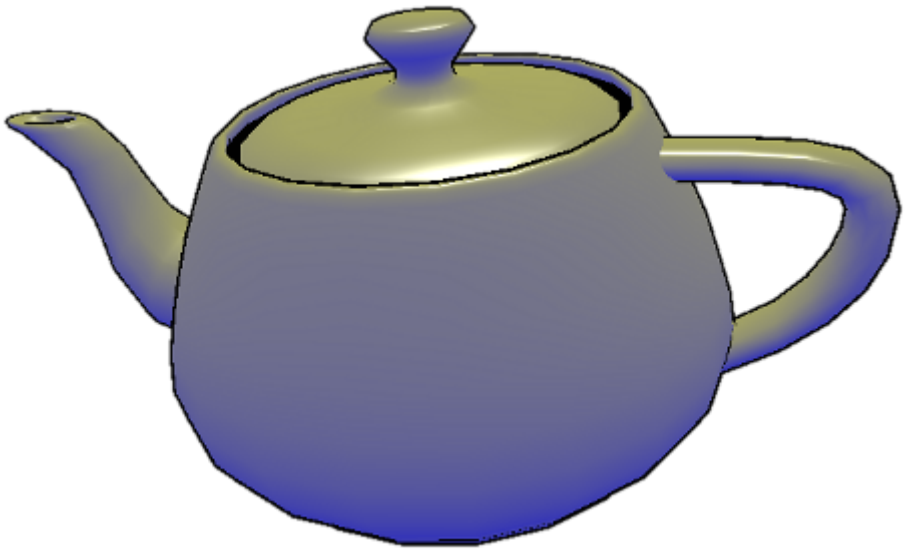
Render Mode

```
    m_goochAppearance1->SetRenderMode(RenderMode::Create());
    m_goochAppearance1->GetRenderMode()->SetCulling(RenderMode::BackFaceCulling);

    m_goochAppearance2->SetRenderMode(RenderMode::Create());
    m_goochAppearance2->GetRenderMode()->SetCulling(RenderMode::FrontFaceCulling);
```

Example

Here is exemplified the Gooch effect:



Gooch Effect in SceneComposer

To experiment with the Gooch effect in SceneComposer use solution **ShaderExamples** from folder *cgi_studio_player/content/Tutorials/04_ShaderUsage*.

Multi Pass Effects in Candera > Examples for Global 3D Effect Shaders

Description

This chapter describes how global multi pass effects in Candera can be achieved and special 3D effect shaders techniques supported by Candera.

Depth of Field

Description

This chapter describes how to generate the **Depth of Field** effect using the global multipass appearance technique supported by Candera.

Introduction

Depth of Field

The Depth-of-Field effect simulates the natural blurring of foreground and background scene elements when viewed through a camera lens. The effect firstly separates the scene in Z order and then blurs the foreground and background images according to the values set in the Depth of Field effect parameters. The final image is composited from the processed originals.

In Candera, the effect is realized using a global multipass technique which implies the rendering in four steps: [1]

- Render pass 1 - rendering the sharp scene, storing depth values in alpha channel
- Render pass 2 - blurring the result of the first render pass horizontally
- Render pass 3 - blurring the result of the second render pass vertically
- Render pass 4 - combine sharp and blurred picture, using focus-, focus range- and stored depth values, in order to realize the final result

Workflow

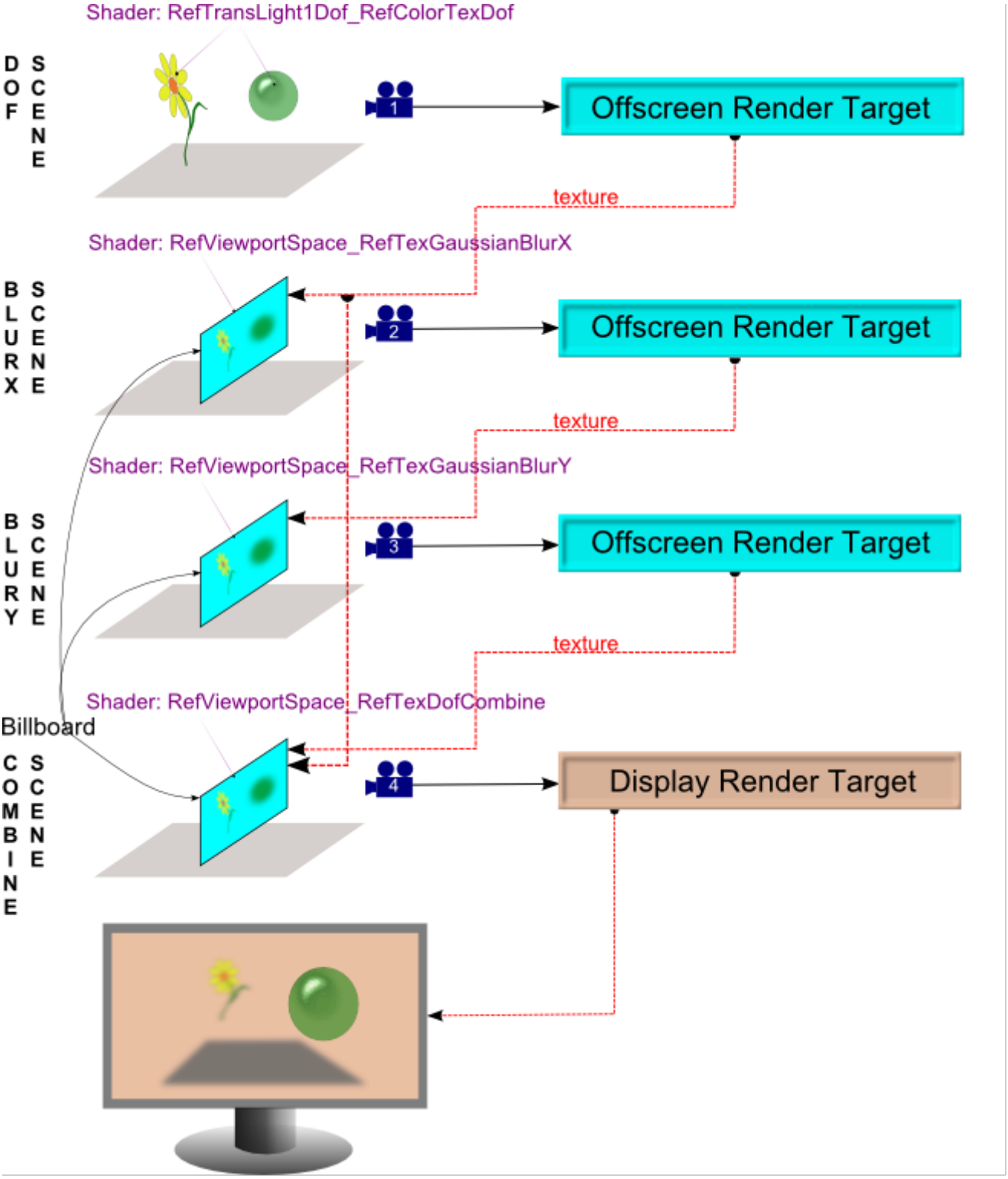
Scenes

For an easier understanding on how to realize the Depth of Field effect we will use four scenes, each scene will correspond to a rendering step.

First of all, the original main scene is taken and used as a texture on a billboard. The depth values of the images are saved in the alpha value in the offscreen frame buffer. In the second and third steps, the blurring is being done. For reasons of the performance, this step is parted into two steps: The resulting billboard of the first workflow step is blurred horizontally using a fragment shader. The resulting image is used as texture input for the next scene, which is subsequently blurred vertically.
In the final step, the blurred images are combined with the sharp images from the first offscreen render target. The result is rendered as texture on a billboard placed in the final scene. A fourth camera displays the resulting Depth of Field effect applied on the main scene.

> The Billboards should be rendered such that they cover the whole screen

The technique is also described schematically in the diagram below: [1]



> Remember to set a Candera::RenderOrder object for each scene.

The sections below offer you some extra information needed to correctly set up the effect.

Offscreen Render Targets

Create and configure all offscreen render targets that shall be used for cameras 1 - 3 as shown below:

```
    m_offscreenRenderTargetDof = Base::CreateGraphicDeviceUnit(DevicePackageDescriptor::Offscree
    if (m_offscreenRenderTargetDof == 0) {
        return false;
    }

    sprintf(m_rtProperties.m_width, "%d", m_gdu->ToRenderTarget3D()->GetWidth());
    sprintf(m_rtProperties.m_height, "%d", m_gdu->ToRenderTarget3D()->GetHeight());
    sprintf(m_rtProperties.m_colorFormat, "%d", RgbaUnpackedColorFormat);
    sprintf(m_rtProperties.m_depthFormat, "%d", Depth32Format);

    const MetaInfo::GraphicDeviceUnitMetaInfo *meta = DevicePackageDescriptor::GetMetaInformatic
    meta->LookupItem("Width") && meta->LookupItem("Width")->Set(m_offscreenRenderTargetDof, m_rt
    meta->LookupItem("Height") && meta->LookupItem("Height")->Set(m_offscreenRenderTargetDof, m_
    meta->LookupItem("ColorFormat") && meta->LookupItem("ColorFormat")->Set(m_offscreenRenderTar
    meta->LookupItem("DepthFormat") && meta->LookupItem("DepthFormat")->Set(m_offscreenRenderTar

#if CANDERA_DEVICE_IMX6
    static_cast<iMX6FrameBufferObject*>(m_offscreenRenderTargetDof)->GetProperties().SetOwner(m_
#endif
    m_offscreenRenderTargetDof->Upload();
```

Shader Parameter Setter

The meshes from each scene will use a dedicated shader which will set specific uniforms for each scene. The uniforms needed to be set are:

- Scaled depth value (u_depthScale): factor used to scale the depth value
- Circle of confusion (u_CocScale): factor used to scale the size of circle of confusion
- Focus range (u_Range): factor used in the computation of the weight of sharpness / blurriness
- Focus (u_Focus): factor used in the computation of the weight of sharpness / blurriness

```
    // DoF Scene
    m_objectSps = GenericShaderParamSetter::Create();
    m_objectSps->SetLightsCoordinateSpace(Light::Object);
    m_objectSps->SetModelMatrix4Enabled(false);
    m_objectSps->SetNormalModelMatrix3Enabled(false);
    m_objectSps->SetUniform("u_depthScale",Shader::Float, &m_depthScale);

    // Blur Scene vertically
    m_blurXSps = GenericShaderParamSetter::Create();
    m_blurXSps->SetModelViewProjectionMatrix4Enabled(false);
    m_blurXSps->SetUniform("u_blurFactor",Shader::Float, &m_blurFactor);
    m_blurXSps->SetUniform("u_offsetLeft", Shader::Float, &m_viewportLeft);
    m_blurXSps->SetUniform("u_offsetTop", Shader::Float, &m_viewportTop);

    // Blur Scene horizontally
```

```
    m_blurYSps = GenericShaderParamSetter::Create();
    m_blurYSps->SetModelViewProjectionMatrix4Enabled(false);
    m_blurYSps->SetUniform("u_blurFactor",Shader::Float, &m_blurFactor);
    m_blurYSps->SetUniform("u_offsetLeft", Shader::Float, &m_viewportLeft);
    m_blurYSps->SetUniform("u_offsetTop", Shader::Float, &m_viewportTop);

    // Combine Scene
    m_dofCombineSps = GenericShaderParamSetter::Create();
    m_dofCombineSps->SetModelViewProjectionMatrix4Enabled(false);
    m_dofCombineSps->SetUniform("u_Range",Shader::Float, &m_range);
    m_dofCombineSps->SetUniform("u_Focus",Shader::Float, &m_focus);
    m_dofCombineSps->SetUniform("u_offsetLeft", Shader::Float, &m_viewportLeft);
    m_dofCombineSps->SetUniform("u_offsetTop", Shader::Float, &m_viewportTop);
```

Camera

The cameras from the all three scenes should use perspective projection.

```
    SharedPointer<PerspectiveProjection> perspectiveProjection = PerspectiveProjection::Create
();
    perspectiveProjection->SetNearZ(0.001F);
    perspectiveProjection->SetFarZ(100.0F);
    perspectiveProjection->SetFovYDegrees(45.0F);
    perspectiveProjection->SetAspectRatio(static_cast<Float>(dispSettings->width) / static_cast<
```

Set the clear mode for the first two cameras as below:

```
    m_clearMode.SetClearColor(Color(0.8f,0.8f,0.8f,1.0f));
    m_clearMode.SetColorClearEnabled(true);
    m_clearMode.SetDepthClearEnabled(true);
    m_clearMode.SetClearDepth(1.0f);
```

Billboard Textures

This code snippet shows you how to create the textures for the billboards. First create a Candera::ProxyTextureImage:

```
    SharedPointer<ProxyTextureImage> proxyTexImgBlurredX = ProxyTextureImage::Create
(m_offscreenRenderTargetBlurX->ToImageSource3D());
    SharedPointer<ProxyTextureImage> proxyTexImgBlurredY = ProxyTextureImage::Create
(m_offscreenRenderTargetBlurY->ToImageSource3D());
```

Then create and configure the texture:

```
    SharedPointer<Texture> proxyTexBlurredX = Texture::Create();
    proxyTexBlurredX->SetTextureImage(proxyTexImgBlurredX);
    proxyTexBlurredX->SetMipMapFilter(Texture::MipMapNone);
    proxyTexBlurredX->SetMagnificationFilter(Texture::MinMagNearest);
    proxyTexBlurredX->SetMinificationFilter(Texture::MinMagNearest);
```

```
proxyTexBlurredX->SetWrapModeU(Texture::ClampToEdge);
proxyTexBlurredX->SetWrapModeV(Texture::ClampToEdge);

SharedPointer<Texture> proxyTexBlurredY = Texture::Create();
proxyTexBlurredY->SetTextureImage(proxyTexImgBlurredY);
proxyTexBlurredY->SetMipMapFilter(Texture::MipMapNone);
proxyTexBlurredY->SetMagnificationFilter(Texture::MinMagNearest);
proxyTexBlurredY->SetMinificationFilter(Texture::MinMagNearest);
proxyTexBlurredY->SetWrapModeU(Texture::ClampToEdge);
proxyTexBlurredY->SetWrapModeV(Texture::ClampToEdge);
```

Repeat these two steps for all textures.

> The billboard from the Combine Scene uses two textures. When you set the textures be sure that the *texture unit* value is set to "0" for the texture corresponding to the camera 1. Analogous, the other texture will have the *texture unit* value set to "1".

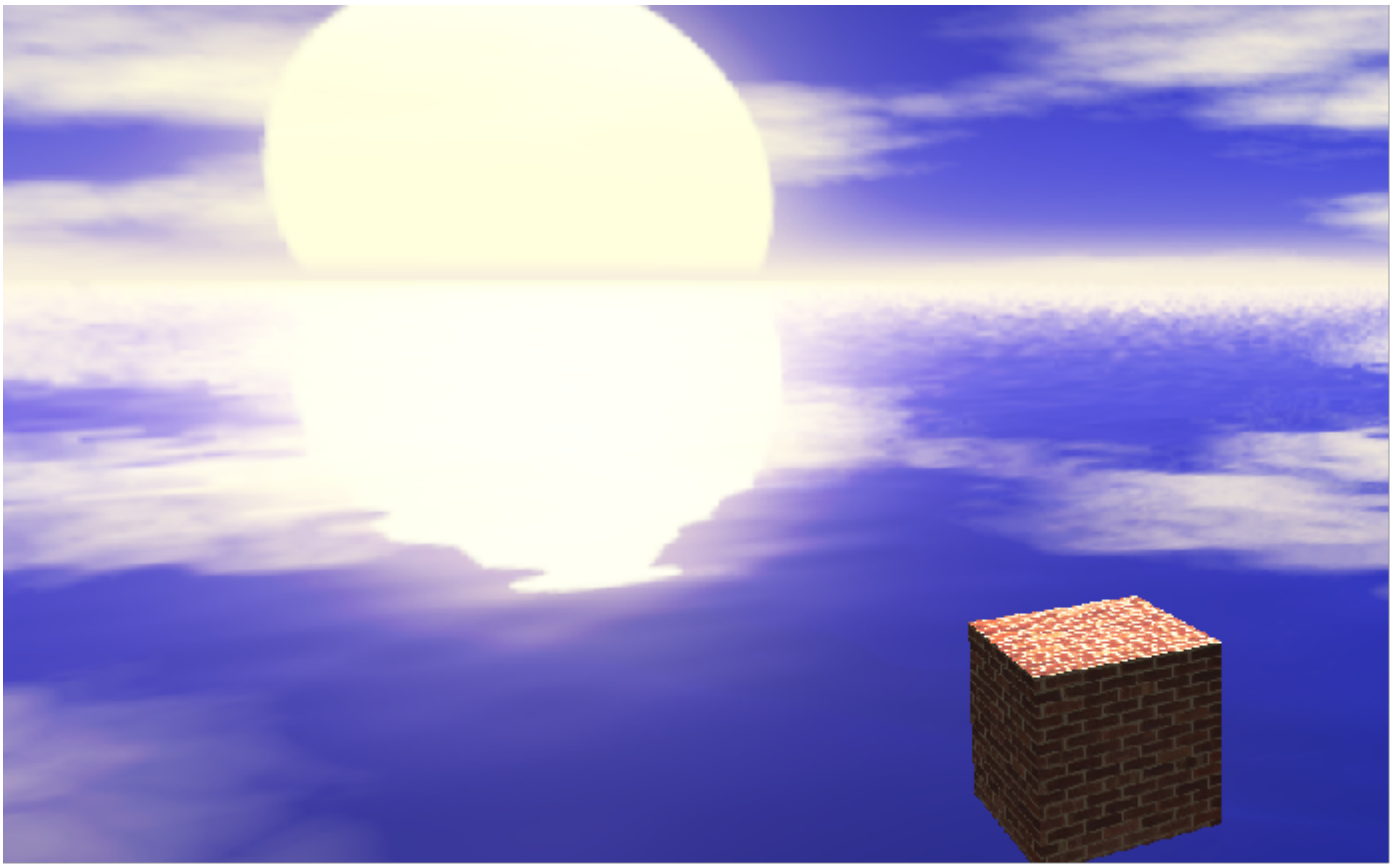
Placing Billboards in Scenes

See Shader Usage in Candera: Global Multi Pass Effects

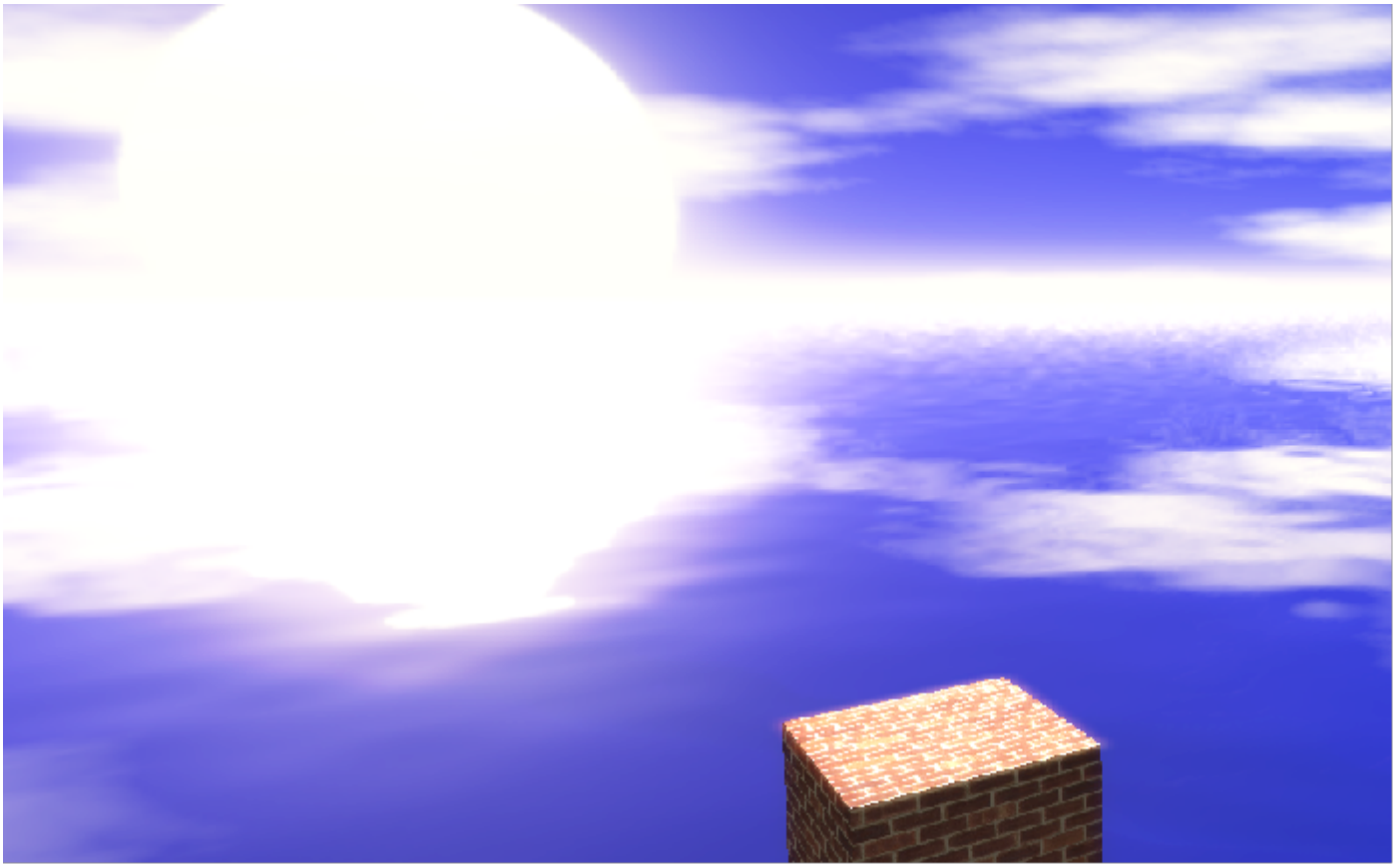
Example

Here is exemplified the **Bloom** effect on a scene.

First image shows the scene without the effect:



Second image shows same scene but with the effect:



References

[1] http://www.geeks3d.com/20090710/javascript-depth-of-field-effect

Bloom

Description

This chapter describes how to generate the **Bloom** effect using the global multipass appearance technique supported by Candera.

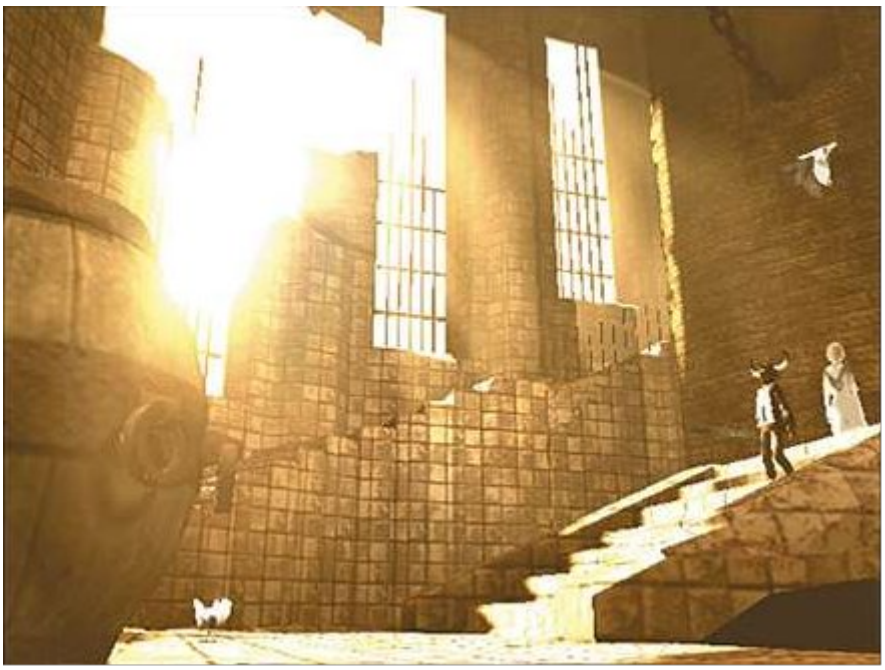
Introduction

Bloom

The Bloom-Effect is a shader effect used to simulate the appearance of very bright light. The effect firstly extracts the bright area of an image using a threshold filter. The extracted regions are further blurred in order to realize the simulation of light bleeding over darker regions.

The bloom effect is realized using a global multipass technique which implies the rendering in five steps: [1]

- Render pass 1 - rendering the sharp scene
- Render pass 2 - extract bright area using a threshold filter
- Render pass 3 - blurring the result of the second render pass horizontally
- Render pass 4 - blurring the result of the third render pass vertically
- Render pass 5 - combine sharp and blurred picture, using intensity as well as saturation of bloom texture and original scene texture, in order to generate the final result



Workflow

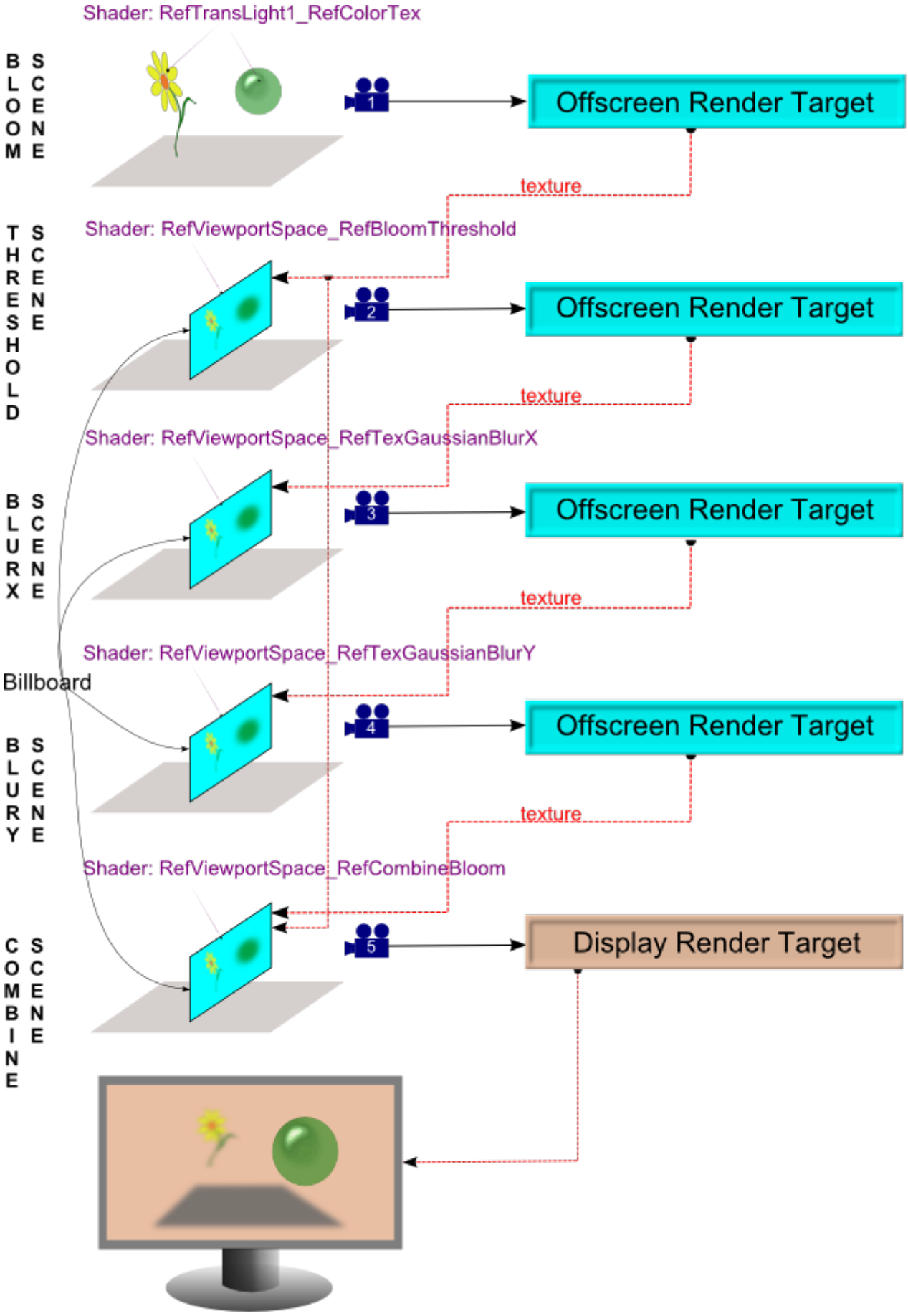
Scenes

For an easier understanding on how to realize the Bloom effect we will use five scenes, each scene will correspond to a rendering step.

First of all, the original main scene is taken and used as a texture on a billboard. This billboard is rendered with a special shader that will extract the bright area of the texture using a threshold filter.
In the second and third steps, the blurring is being done. For reasons of the performance, this step is parted into two steps: The resulting billboard of the first workflow step is rendered with a shader that will blur the images horizontally and these blurred images are rendered as texture on an another billboard placed in the third scene. The billboard of the third scene is then rendered with a shader that will blur the images vertically. This billboard is part of the fourth scene, viewed by a camera which is connected to a fourth offscreen render target.
In the final step, the images of the blurred, bright area of the scene are combined with the sharp image from the first offscreen render target. The result is rendered as texture on a billboard placed in the final fifth scene. A fifth camera displays the resulting Bloom effect applied on the main scene.

> The Billboards should be rendered such that they cover the whole screen

The technique is also described schematically in the diagram below:


Shader: RefTransLight1_RefColorTex
BLOOM SCENE
1
Offscreen Render Target
texture
Shader: RefViewportSpace_RefBloomThreshold
THRESHOLD SCENE
2
Offscreen Render Target
texture
Shader: RefViewportSpace_RefTexGaussianBlurX
BLURX SCENE
3
Offscreen Render Target
texture
Shader: RefViewportSpace_RefTexGaussianBlurY
Billboard
BLURY SCENE
4
Offscreen Render Target
texture
Shader: RefViewportSpace_RefCombineBloom
COMBINE SCENE
5
Display Render Target

> Remember to set aCandera::RenderOrder object for each scene.

The sections below offer you some extra information needed to correctly set up the effect.

Shader Parameter Setter

The meshes from each scene will use a dedicated shader which will set specific uniforms for each scene. The uniforms needed to be set are:

- Bloom Brightness Threshold value (u_brightnessThreshold): brightness threshold value to render the bloom shape. lower = darker parts of the image
- Blur Factor (u_blurFactor): factor used in the computation of the weight of sharpness / blurriness
- Intensity amount on original scene (u_originalIntensity): factor used to set the intensity of the original scene texture
- Saturation amount on original scene (u_originalSaturation): factor used to set the saturation of the original scene texture
- Bloom Intensity (u_bloomIntensity): bloom intensity factor used in the combining scene
- Bloom Saturation (u_bloomSaturation): bloom saturation value used in the combining scene

```
// Sharp Scene
m_objectSps = GenericShaderParamSetter::Create();
m_objectSps->SetLightsCoordinateSpace(Light::Object);
m_objectSps->SetModelMatrix4Enabled(false);
m_objectSps->SetNormalModelMatrix3Enabled(false);

// Bloom Threshold Scene
m_bloomThresholdSps = GenericShaderParamSetter::Create();
m_bloomThresholdSps->SetModelViewProjectionMatrix4Enabled(false);
m_bloomThresholdSps->SetUniform("u_brightnessThreshold",Shader::Float, &m_bloomBrightnessThr
m_bloomThresholdSps->SetUniform("u_offsetLeft", Shader::Float, &m_offsetLeft);
m_bloomThresholdSps->SetUniform("u_offsetTop", Shader::Float, &m_offsetTop);

// Blur Scene vertically
m_bloomBlurXSps = GenericShaderParamSetter::Create();
m_bloomBlurXSps->SetModelViewProjectionMatrix4Enabled(false);
m_blurFactor = 1.0F / m_gdu->ToRenderTarget3D()->GetWidth();
m_bloomBlurXSps->SetUniform("u_blurFactor",Shader::Float, &m_blurFactor);
m_bloomBlurXSps->SetUniform("u_offsetLeft", Shader::Float, &m_offsetLeft);
m_bloomBlurXSps->SetUniform("u_offsetTop", Shader::Float, &m_offsetTop);

// Blur Scene horizontally
m_bloomBlurYSps = GenericShaderParamSetter::Create();
m_bloomBlurYSps->SetModelViewProjectionMatrix4Enabled(false);
m_blurFactor = 1.0F / m_gdu->ToRenderTarget3D()->GetHeight();
m_bloomBlurYSps->SetUniform("u_blurFactor",Shader::Float, &m_blurFactor);
```

```
    m_bloomBlurYSps->SetUniform("u_offsetLeft", Shader::Float, &m_offsetLeft);
    m_bloomBlurYSps->SetUniform("u_offsetTop", Shader::Float, &m_offsetTop);

    // Combine Scene
    m_bloomCombineSps = GenericShaderParamSetter::Create();
    m_bloomCombineSps->SetModelViewProjectionMatrix4Enabled(false);
    m_bloomCombineSps->SetUniform("u_originalIntensity",Shader::Float, &m_originalIntensity);
    m_bloomCombineSps->SetUniform("u_originalSaturation",Shader::Float, &m_originialSaturation);
    m_bloomCombineSps->SetUniform("u_bloomIntensity",Shader::Float, &m_bloomIntensity);
    m_bloomCombineSps->SetUniform("u_bloomSaturation",Shader::Float, &m_bloomSaturation);
    m_bloomCombineSps->SetUniform("u_offsetLeft",Shader::Float, &m_offsetLeft);
    m_bloomCombineSps->SetUniform("u_offsetTop",Shader::Float, &m_offsetTop);
```

Camera

The cameras from all the five scenes should use perspective projection.

```
    SharedPointer<PerspectiveProjection> perspectiveProjection = PerspectiveProjection::Create
();
    perspectiveProjection->SetNearZ(0.001F);
    perspectiveProjection->SetFarZ(100.0F);
    perspectiveProjection->SetFovYDegrees(45.0F);
    perspectiveProjection->SetAspectRatio(static_cast<Float>(dispSettings->width) / static_cast<
```

Set the clear mode for the cameras as below:

```
    m_clearMode.SetClearColor(Color(0.8f,0.8f,0.8f,1.0f));
    m_clearMode.SetColorClearEnabled(true);
    m_clearMode.SetDepthClearEnabled(true);
    m_clearMode.SetClearDepth(1.0f);
```

Billboard Textures

This code snippet shows you how to create the textures for the billboards. First create a Candera::ProxyTextureImage:

```
    SharedPointer<ProxyTextureImage> proxyTexImgBloomThreshold = ProxyTextureImage::Create
(m_offscreenRenderTargetThreshold->ToImageSource3D());
    SharedPointer<ProxyTextureImage> proxyTexImgBlurredX = ProxyTextureImage::Create
(m_offscreenRenderTargetBlurX->ToImageSource3D());
    SharedPointer<ProxyTextureImage> proxyTexImgBlurredY = ProxyTextureImage::Create
(m_offscreenRenderTargetBlurY->ToImageSource3D());
```

Then create and configure the texture:

```
    SharedPointer<Texture> proxyTexBloomThreshold = Texture::Create();
    proxyTexBloomThreshold->SetTextureImage(proxyTexImgBloomThreshold);
    proxyTexBloomThreshold->SetMipMapFilter(Texture::MipMapNone);
    proxyTexBloomThreshold->SetMagnificationFilter(Texture::MinMagNearest);
```

```
proxyTexBloomThreshold->SetMinificationFilter(Texture::MinMagNearest);
proxyTexBloomThreshold->SetWrapModeU(Texture::ClampToEdge);
proxyTexBloomThreshold->SetWrapModeV(Texture::ClampToEdge);

SharedPointer<Texture> proxyTexBlurredX = Texture::Create();
proxyTexBlurredX->SetTextureImage(proxyTexImgBlurredX);
proxyTexBlurredX->SetMipMapFilter(Texture::MipMapNone);
proxyTexBlurredX->SetMagnificationFilter(Texture::MinMagNearest);
proxyTexBlurredX->SetMinificationFilter(Texture::MinMagNearest);
proxyTexBlurredX->SetWrapModeU(Texture::ClampToEdge);
proxyTexBlurredX->SetWrapModeV(Texture::ClampToEdge);

SharedPointer<Texture> proxyTexBlurredY = Texture::Create();
proxyTexBlurredY->SetTextureImage(proxyTexImgBlurredY);
proxyTexBlurredY->SetMipMapFilter(Texture::MipMapNone);
proxyTexBlurredY->SetMagnificationFilter(Texture::MinMagNearest);
proxyTexBlurredY->SetMinificationFilter(Texture::MinMagNearest);
proxyTexBlurredY->SetWrapModeU(Texture::ClampToEdge);
proxyTexBlurredY->SetWrapModeV(Texture::ClampToEdge);
```

> The billboard from the Combine Scene uses two textures. When you set the textures be sure that the *texture unit* value is set to "0" for the texture corresponding to the camera 1. Analogous, the other texture will have the *texture unit* value set to "1".

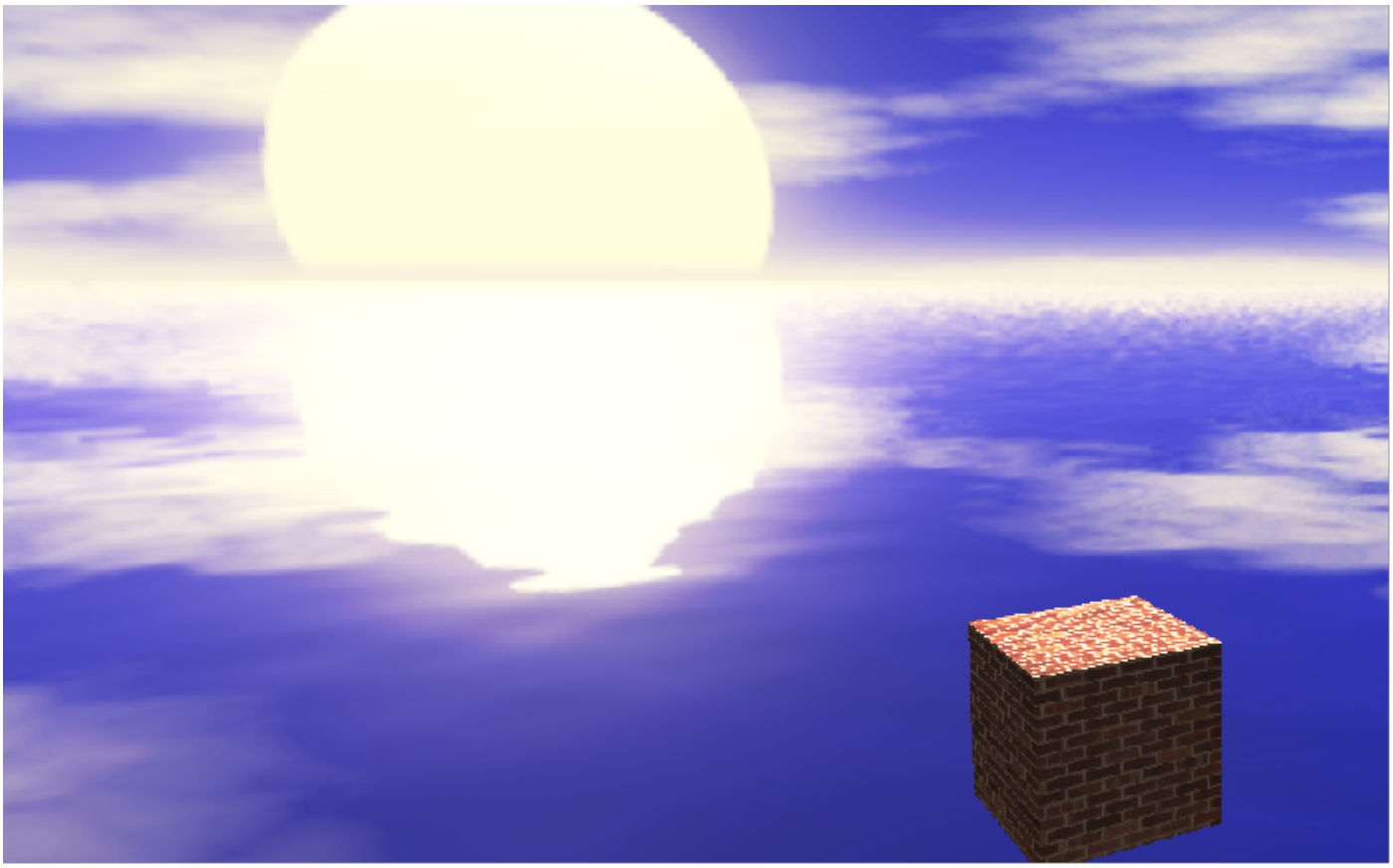
Placing Billboards in Scenes

See Shader Usage in Candera: Global Multi Pass Effects

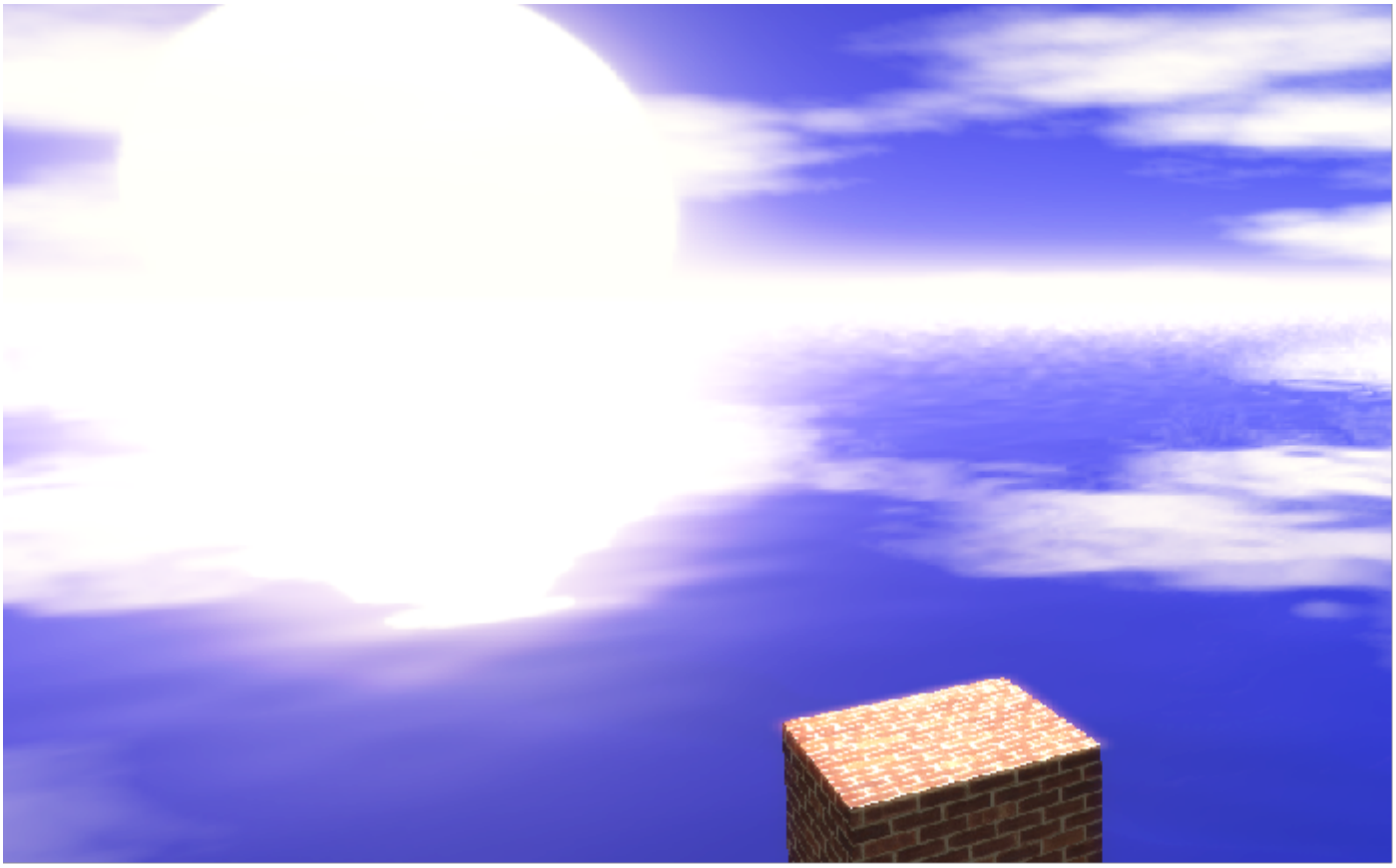
Example

Here is exemplified the **Bloom** effect on a scene.

First image shows the scene without the effect:



Second image shows same scene but with the effect:

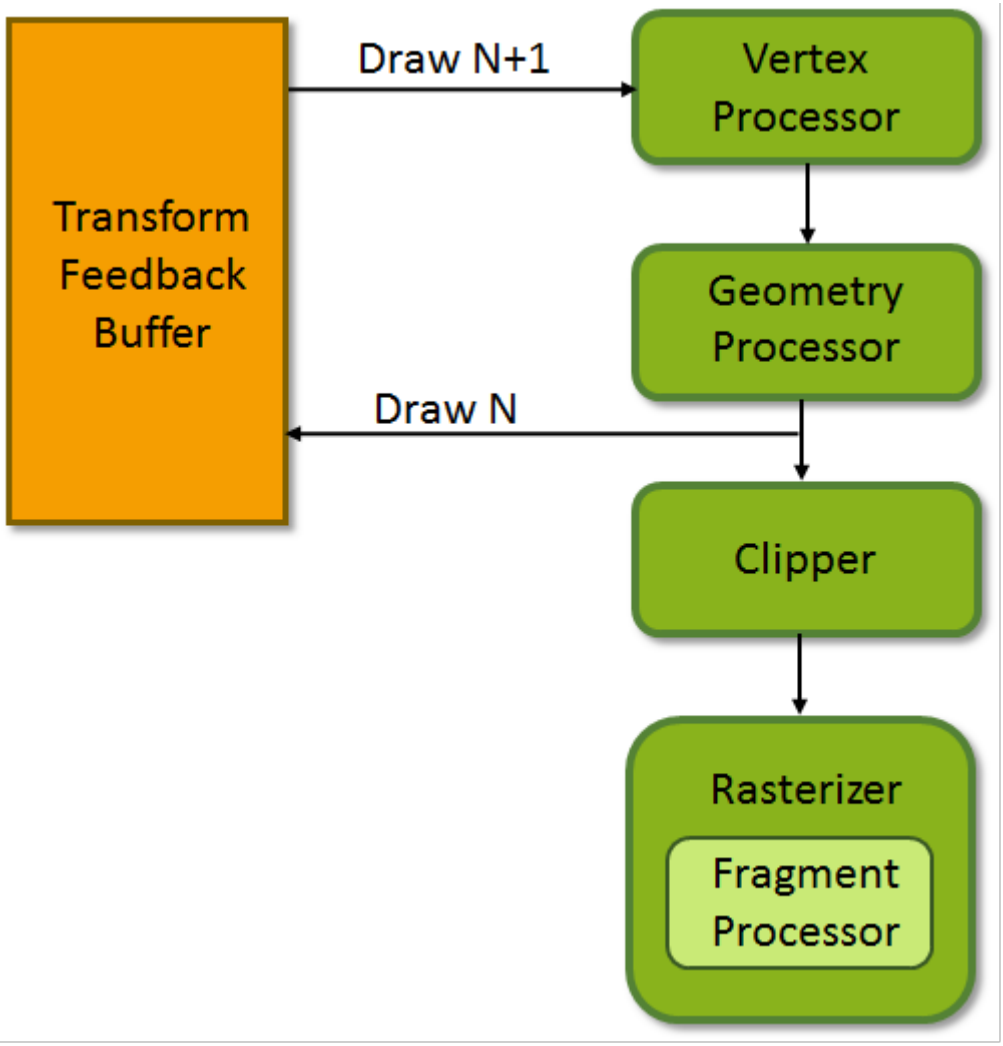


Transform Feedback

Transform Feedback is the process of capturing primitives generated by the vertex processing step(s), recording data from those primitives into a vertex buffer. This allows one to preserve the post-transform rendering state of an object and resubmit this data multiple times.

Use Cases

This feature enables the user to utilize the vertex processing unit of the GPU to perform calculations and reuse the results. This is especially useful for debugging vertex transformations. Applications can also use it for use cases like speeding up particle systems.



Usage in Candera:

1. Before uploading your vertex shader that processes your data, use Shader::SetTransformVaryings to tell the engine which shader varying(s) need to be recorded into the feedback buffer.

2. Upload the shader, activate the shader’s appearance, and source vertex buffer as usual. If you want to disable the output of the pixel pipeline stage, use RenderMode::SetRasterizerDiscardEnabled.
3. Bind the source vertex buffer using Shader::BindAttributes.
4. Call Renderer::BindTransformFeedbackBuffer, and Renderer::BeginTransformFeedbackBuffer using the destination vertex buffer that the processed primitives will be recorded into.
5. Render the source vertex buffer as usual.
6. Call Renderer::EndTransformFeedbackBuffer
7. Call Renderer::MapTransformFeedbackBuffer to get a pointer to the processed, and recorded data.
8. Before using that pointer for your own purposes, call Renderer::UnmapTransformFeedbackBuffer.
9. The lifetime of that pointer’s data is tied to the lifetime of your destination vertex buffer.