Tutorial for Graphic Device Interface

Cameras and Render Targets

Description

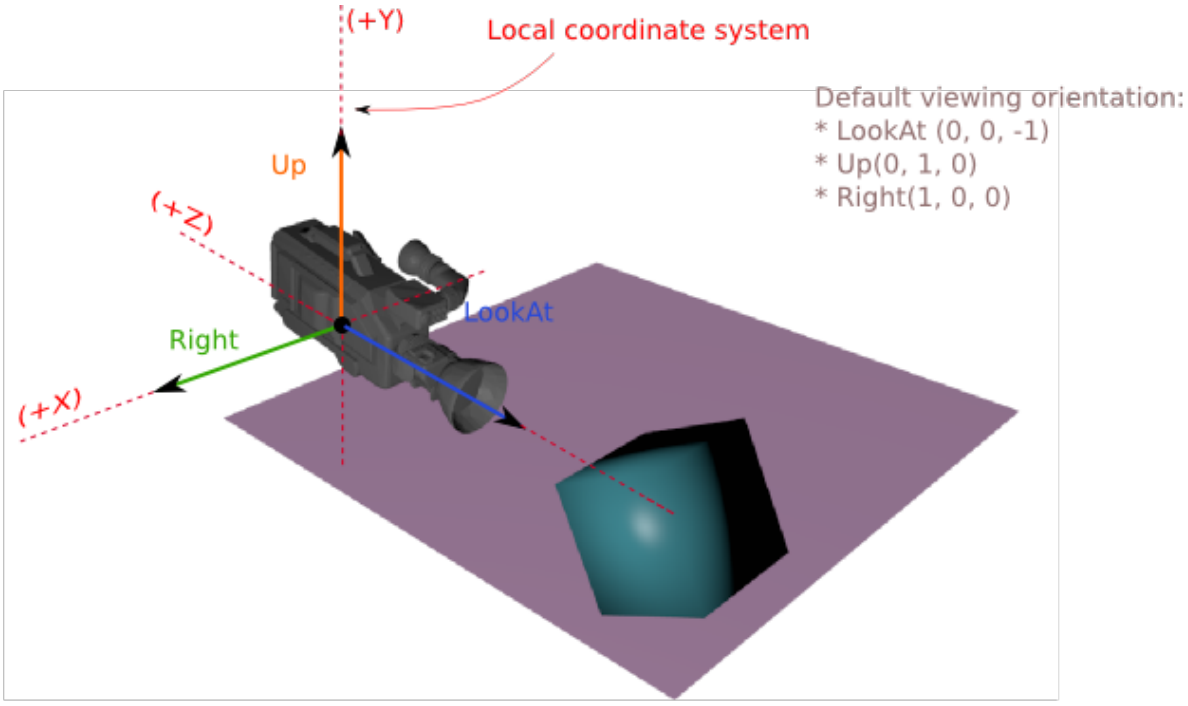
Content specified in a Candera scene graph is rendered by configuring at least one camera on it. The render result of the camera is projected on a so called render target.

This chapter gives an overview about cameras and render targets in Candera.

Go to EGL Context Handling to skip the introduction and proceed with the first example for this topic.

Camera Orientation

- modifying the "rotation" part of Transformable (Candera::Transformable::SetRotation),
- setting the two orientation vectors:
  - the local "LookAt" vector which can be set by the functions Candera::Camera::SetLookAtVector, Candera::Camera::LookAtWorldPoint, Candera::Camera::RotateAroundWorldAxis
  - the local "Up" vector, set by the function Candera::Camera::SetUpVector



- setting a "LookAt-Node." If such a node is set the camera will always look at that node. This behavior overrides other rotation transformations set by the methods mentioned above. The node can be set by the function Candera::Camera::SetLookAtNode.

The viewing orientation of the Candera::Camera2D can be controlled only by modifying the local rotation using the Candera::Transformable2D::SetRotation function.

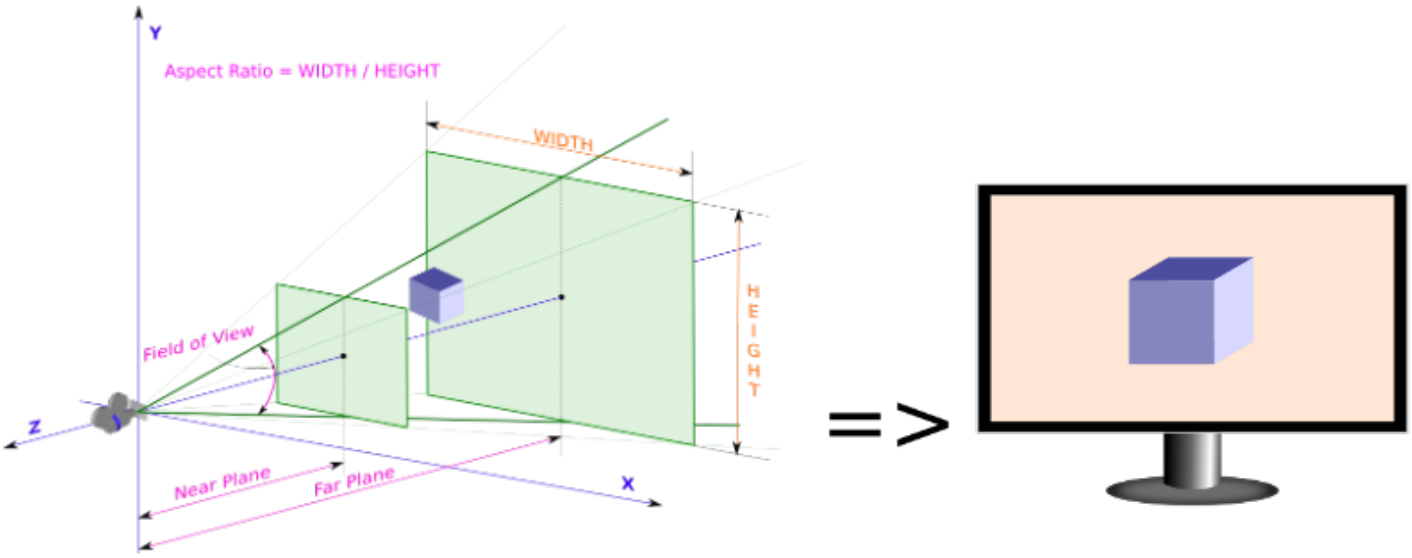
See also:

- Camera Gizmos in SceneComposer User Manual

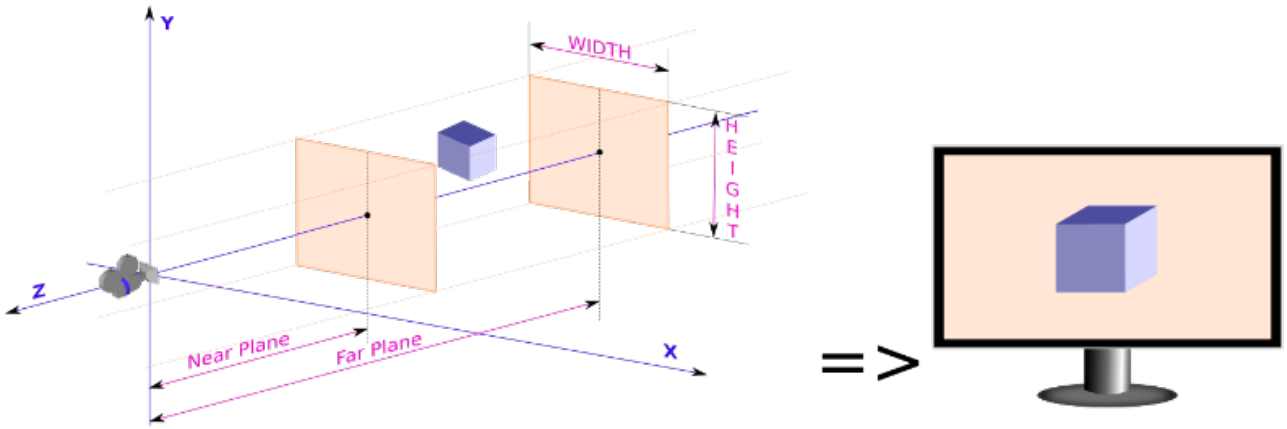
Camera Projections

Candera::Camera::SetProjection is analogous to choosing a lens for the camera.

- Candera::PerspectiveProjection: This is the most commonly used projection. It provides proper visual effects depending on the distance between 3D objects and the camera projecting them. Specify the field of view (FoV) in degrees and the aspect ratio to adjust the camera lens properly.



- Candera::OrthographicProjection: This tutorial chapter has a use case where an OrthographicProjection is needed instead of a PerspectiveProjection. This projection type provides an orthogonal view on the projection plane.

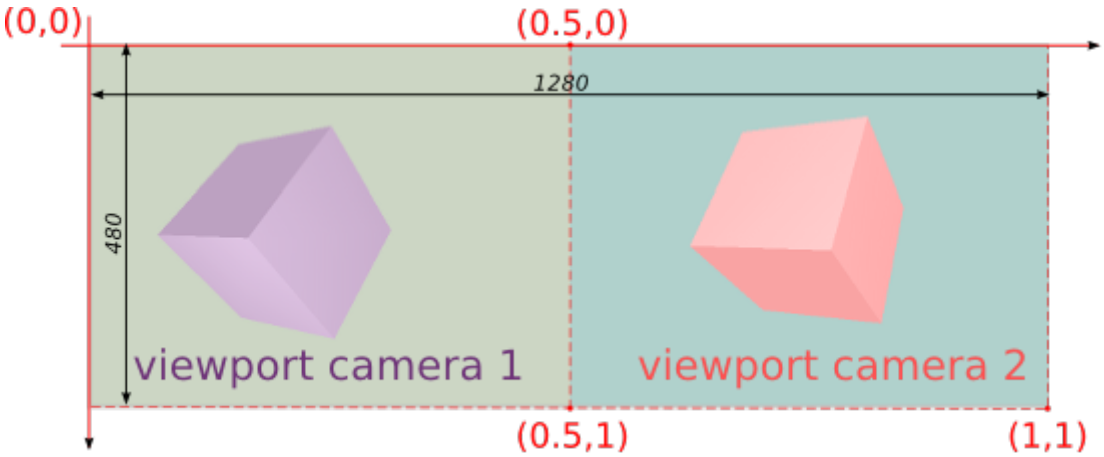


- Candera::GenericProjection is an option to specify an arbitrary projection matrix.

Camera Viewport

Use Candera::Camera::SetViewport to specify a rectangular region on the render target for projecting the camera output. The position and size of the viewport should be given as normalized screen coordinates [0..1]. If the viewport aspect ratio differs from the aspect ratio of the Camera's projection matrix, the final render result will appear stretched or compressed. The default value for viewport to cover the complete render target area is: left 0, right 0, width 1, height 1.

The example below shows how to set the viewport in case of two cameras connected to the same display render target:




```
    perspectiveProjection->SetAspectRatio(1.3333f)); // (1280/2)/480
    ...
    m_camera1 = Camera::Create();
    m_camera1->SetViewport(Candera::Rectangle(0.0f, 0.0f, 0.5f, 1.0f));
```

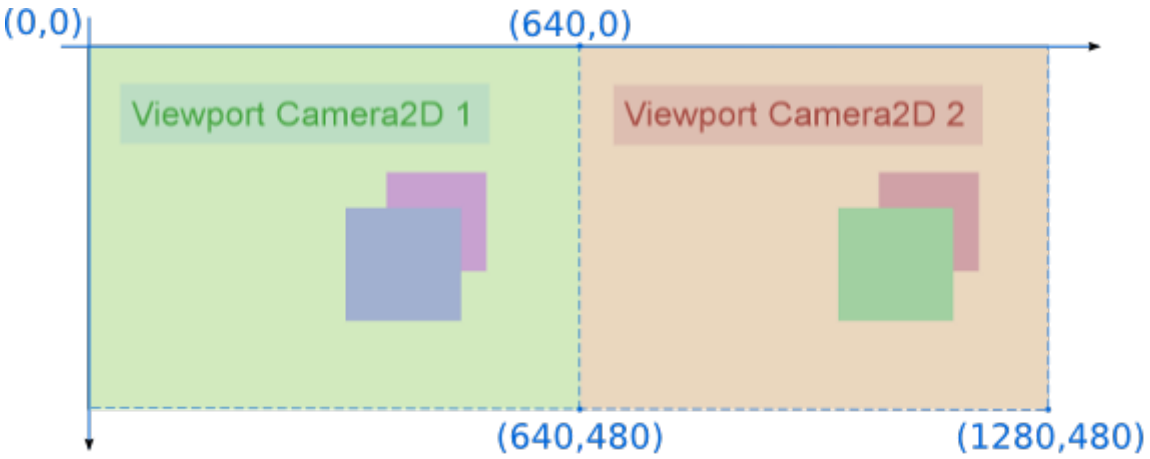
```
m_camera1->SetRenderTarget(m_gdu->ToRenderTarget3D());
m_camera1->SetProjection(perspectiveProjection);
m_camera1->SetClearMode(m_clearMode1);
m_camera1->SetClearModeEnabled(true);
m_camera1->SetSwapEnabled(true);

m_camera2 = Camera::Create();
m_camera2->SetViewport(Candera::Rectangle(0.5f, 0.0f, 0.5f, 1.0f));
m_camera2->SetRenderTarget(m_gdu->ToRenderTarget3D());
m_camera2->SetProjection(perspectiveProjection);
m_camera2->SetClearMode(m_clearMode2);
m_camera2->SetClearModeEnabled(true);
m_camera2->SetSwapEnabled(false)
```

Candera::Camera2D::SetViewport works similarly but in 2D context. The default value for a viewport of a 2D camera are:

- left: 0
- right: 0
- width: -1
- height: -1

> The viewport position and size of a Candera::Camera2D are given in pixels.

Below is presented an example similar with the one above but adapted for 2D.




```
m_camera2D_1 = Camera2D::Create();
m_camera2D_1->SetViewport(Candera::Rectangle(0.0f, 0.0f, 640.0f, 480.0f));
m_camera2D_1->SetRenderTarget(m_gdu->ToRenderTarget2D());
m_camera2D_1->SetClearColor(Color(0.8f, 0.9f, 0.7f, 0.1f));
```

```
m_camera2D_1->SetClearColorEnabled((true);
m_camera2D_1->SetSwapEnabled(true);

m_camera2D_2 = Camera2D::Create();
m_camera2D_2->SetViewport(Candera::Rectangle(640.0f, 0.0f, 640.0f, 480.0f));
m_camera2D_2->SetRenderTarget(m_gdu->ToRenderTarget2D());
m_camera2D_2->SetClearColor(Color(0.9f, 0.8f, 0.7f, 0.1f));
m_camera2D_2->SetClearColorEnabled((true);
m_camera2D_2->SetSwapEnabled(false);
```

Render Targets

A render target is a buffer where the video card draws pixels for a scene that is being rendered. Consequently, once a camera is configured, it must be linked to a render target to specify the target for the camera output.

Display and Offscreen Render Targets

There are basically two different types of render targets (Candera::GraphicDeviceUnit) available in Candera, which can render only 2D content, only 3D content or mixed 2D/3D content:

- **Display/Onscreen Render Targets**: A frame buffer is used as render target and directly linked to a Display.
  - DisplayTarget3D
  - DisplayTarget2D
  - Mixed2D3DDisplayTarget
- **Offscreen Render Targets**: A frame buffer is used as render target and can be used as image source in the same or another scene.
  - OffscreenTarget3D
  - OffscreenTarget2D
  - OffscreenTarget2Dto3D
  - OffscreenTarget3Dto2D
  - Mixed2D3DOffscreenTarget

For **iMX6** device there is one more type of render target: Candera:SoftwareLayer. More details about this can be found in Software Layers chapter.

Device Capabilities

For an overview about what kind of render targets are supported on which device please refer to: Device Specific Documentation

Following is an overview about what kind of render targets are supported on which device:

Device	Number of Displays Supported	Types of Layers Supported	Unit Types

See also:

- Link Camera to Render Target in SceneComposer User Manual

Render Target Lifetime

- **Create:** A graphic device unit (GDU) which hosts a render target and/or an image source can be created with function DevicePackageInterface::CreateGraphicDeviceUnit.
- **Upload:** Subsequently, in order to utilize GDU an upload to video memory is required, which is processed by calling function GraphicDeviceUnit::Upload on GDU object.
- **Destroy:** In order to destroy video and system memory resources associated to GDU call functions GraphicDeviceUnit::Unload and DevicePackageInterface::DestroyGraphicDeviceUnit in exactly that order.

In general, follow the sequence below to upload and unload GDUs:

- Construction:
  - 1.) Upload all displays
  - 2.) Upload all onscreen framebuffers (WindowSurface)
  - 3.) Upload all offscreen framebuffers (FramebufferBufferObject)

- Destruction:
  - 1.) Unload all offscreen framebuffers (FramebufferBufferObject)
  - 2.) Unload all onscreen framebuffers (WindowSurface)
  - 3.) Unload all displays

Render Buffer Swapping and Clearing

Render Buffer Swapping

If the draw target operates on multiple buffers after each rendering cycle the presentation buffer is exchanged with the current background buffer, on which was written during the cycle.

For this, three interfaces are available:

- Candera::RenderTarget::SetAutoSwapEnabled(bool enabled): if *true*, render target buffers are swapped automatically, when a camera within the render target has been rendered. Default: *false*
- Candera::Camera::SetSwapEnabled(bool enable): if *true*, buffers are swapped automatically after rendering the according camera. Default: *false*

- Candera::RenderTarget::SwapBuffers(): Trigger buffer swapping 'by hand'

It is very important to control buffer swapping in the application correctly:

- If swapping is never done, the display will stay black
- If buffers are swapped too often, the display will flicker.

The application doesn't need to swap its render targets in case:

- the render target is set to swap automatically. For this, set the render target property "SetAutoSwapEnabled" to *true*.
- the proper camera is set to swap automatically. For this, set the camera property "SwapEnabled" to *true*.

> If you use more than one camera on a render target, take care to enable swapping only for the last camera (highest sequence number).

Render Buffer Clearing

The render buffer can be cleared by one of the cameras attached to the render target or by the render target itself by the means of a shared clear mode object.

Render Buffer Clearing by Camera

Camera2D interface offers two methods related to render buffer clearing. First one, Candera::Camera2D::SetClearColor(), sets the clear color and the other one, Candera::Camera2D::SetClearColorEnabled(), sets whether the clear is performed or not at each render call.

```
camera2D->SetClearColor(Candera::Color(0.5f, 0.5f, 1.0f, 1.0f));
camera2D->SetClearColorEnabled(true);
```

Comparing to the 2D context where only the color buffer needs to be cleared, in 3D, Camera has to clear three buffers: color, depth and stencil. Camera performs the clearing using a Candera::ClearMode object which is attached to it using the method Candera::Camera::SetClearMode().

```
Candera::ClearMode clearmode;
clearmode.SetClearColor(Candera::Color(0.5f, 1.0f, 0.5f, 1.0f));
clearmode.SetColorClearEnabled(true);
```

```
clearmode.SetClearDepth(1.0f);
clearmode.SetDepthClearEnabled(true);

camera3D->SetClearMode(clearmode);
```

> Candera::Camera and Candera::Camera2D clear only the area specified by the viewport.

Render Buffer Clearing by Render Target

The render buffer clearing using the render target is useful especially when the camera viewport is not covering the entire render target. The 2D render target clears the color buffer using a shared clear mode object as shown in the example below:

```
Candera::SharedClearMode2D::SharedPointer shClearMode2D = SharedClearMode2D::Create();
shClearModem2D->GetClearColor() = Candera::Color(0.0f, 0.5f, 0.5f, 1.0f);
m_gdu2D->ToRenderTarget2D()->SetAutoClearEnabled(true);
m_gdu2D->ToRenderTarget2D()->SetClearMode(shClearModem2D);
```

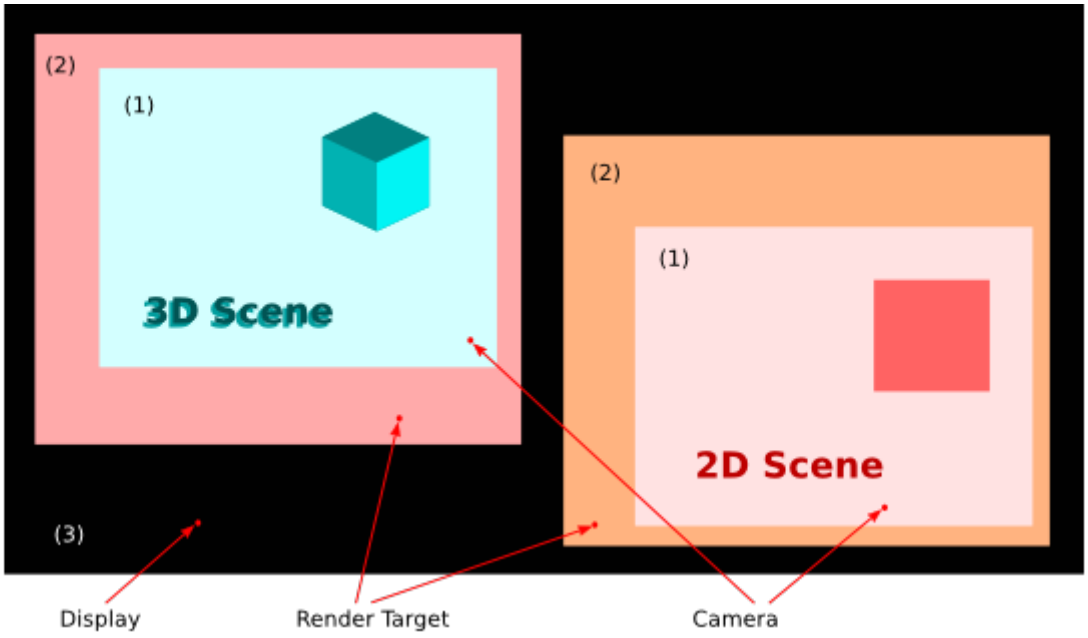
In 3D, the clearing it realized in a similar way but the render target will use a shared clear mode object which has attached a clear mode object.

```
Candera::ClearMode clearmode;
clearmode.SetClearColor(Candera::Color(0.8f, 0.4f, 0.4f, 1.0f));
clearmode.SetColorClearEnabled(true);
clearmode.SetClearDepth(1.0f);
clearmode.SetDepthClearEnabled(true);

Candera::SharedClearMode::SharedPointer shClearMode = Candera::SharedClearMode::Create();
shClearMode->SetClearMode(&clearmode);

gdu->ToRenderTarget3D()->SetClearMode(shClearMode);
gdu->ToRenderTarget3D()->SetAutoClearEnabled(true);
```

Example

Consider two cameras & render targets, 2D and 3D, connected to a common display (see picture below). The clearing is enabled on both cameras and render targets, while clear colors for the render targets and cameras are all different.

- Background color of area (1) is set by the clear color of Camera2D respectively the 3D Camera in the 3D Scene, since the clearing setting of the camera overrides the one from the render target.
- Since the camera viewport is smaller than the render target size, area (2) is cleared by the render target.
- Area (3) cannot be cleared because it is not covered by any of the two render targets.



SceneComposer Example Solution

In SceneComposer, the configuration of render target clearing in 2D context is done similarly to the one of a 2D camera: select the Render Target and, in the Properties panel, enable the "Auto Clear Enabled 2D" property and set the clear color by modifying the "Clear Mode 2D" property.

In 3D context, the activation of auto-clearing on a render target implies to create first a ClearMode: in the Solution Explorer, right click on any of the folders from "Solution" and choose "Add->New Item...->ClearMode". In the "Add new item" dialog enable and set an appropriate value for the Color Clear, Depth Clear and Stencil Clear and then press "OK". Now, select the 3D Render Target and, in the Properties panel, set "Auto Clear Enabled" property to "True" and then click on the "Clear Mode" property and select the previously created ClearMode. An example of usage the render target auto-clear feature in SceneComposer is presented in the solution RenderTargetAutoClearSolution example in folder cgi_studio_player/content/Tutorials/07_RenderTargetsCameras.

Software Layers

Overview

The Candera::SoftwareLayer is a special mixed graphic device unit supported only on specific device packages. This graphic device unit, despite the fact it is a texture render target, it is not meant to be used as Image source in a texture but rather it acts like a display render target by the means of a composition camera (see diagram below).

> Software Layers are not supported on all devices! Software Layers are supported on iMX6. An example solution for this concept can be seen in *cgi_studio_player/content/Tutorials/07_RenderTargetsCameras* solution.

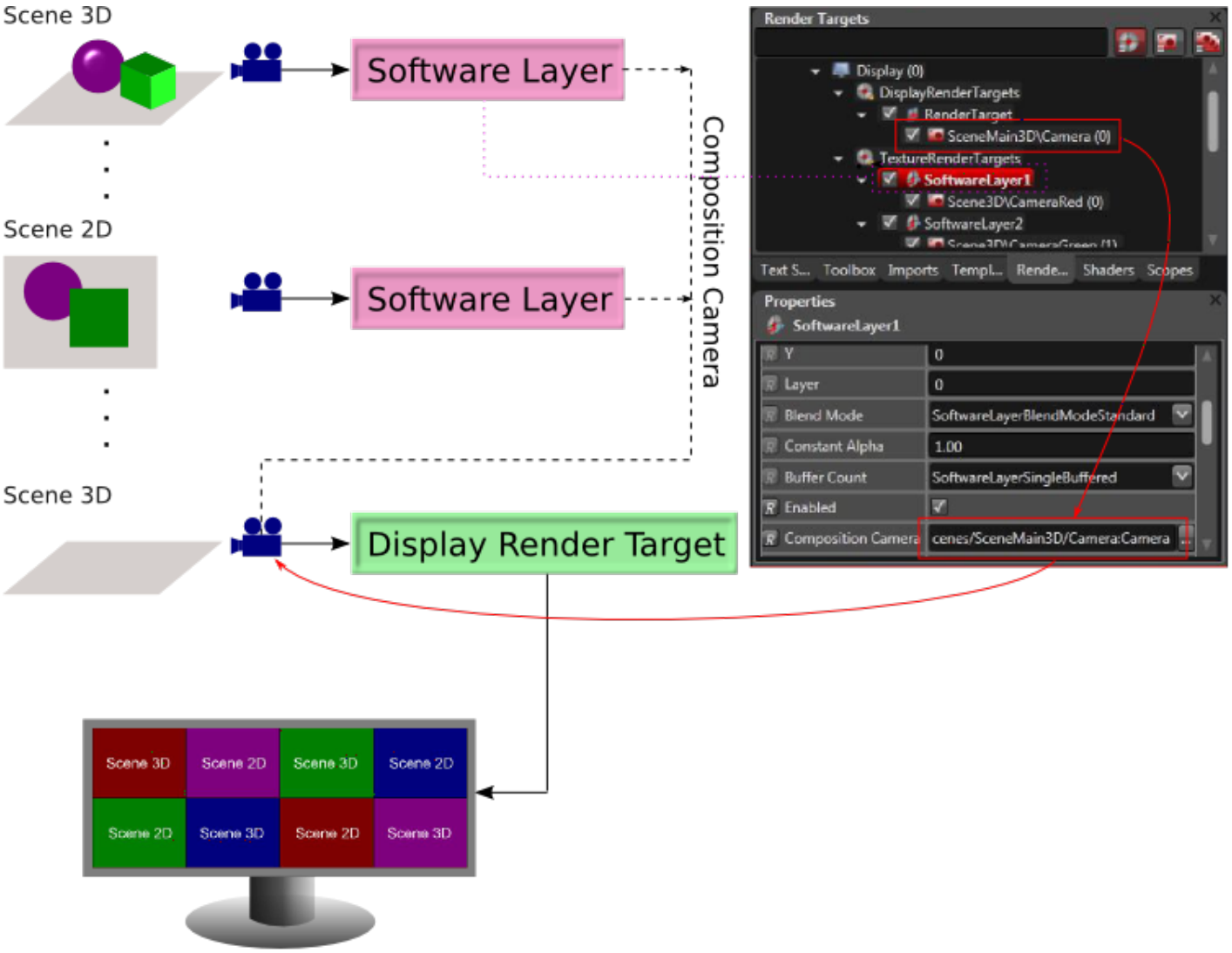
Configuration

> For a correct behaviour of the scene editor the composition camera has to be placed in a separate scene.

The number of software layers which can be blended in one pass is limited by the number of texture units specific for the device.

- For specific devices the maximum number of software layers is 8.

Each of these virtual layers can have a distinct blending mode, buffer settings, size or position, but all have to use the same composition camera, the Candera::Camera which is linked to the display render target:



> The software layers are delimited by a 1 px border. If needed, this border can be disabled using the "BorderEnabled" property.

Layer Order

The software layers are ordered depending on the value set for the Layer property, bigger values meaning that the layer will be closest to the composition camera rendering the main scene.

Blend Modes

Following blend modes are supported for Software Layers:

Source	Destination	Constant Alpha	Blend Mode	Formula	Output
		0.5	SoftwareLayerBlendModeStandard	SRC * SRC.A + DST * (1 - SRC.A)	

SoftwareLayerBlendModePreMultiplied	SRC + DST * (1 - SRC.A)	
SoftwareLayerBlendModeConstantAlpha	SRC * CNS.A + DST * (1 - CNS.A)	
SoftwareLayerBlendModeSimultaneousAlpha	SRC * SRC.A * CNS.A + DST * (1 - SRC.A * CNS.A)	
SoftwareLayerBlendModeSimultaneousPreMultiplied	SRC * CNS.A + DST * (1 - SRC.A * CNS.A)	
SoftwareLayerBlendModeOff	SRC	

For more information about alpha blending please read also the Blending chapther.

Insides

As any other graphic device unit the Candera::SoftwareLayer can be created via Candera::DevicePackageInterface::CreateGraphicDeviceUnit.

- Use Candera::DevicePackageDescriptor::GetUnitTypes to get its type handle as a second parameter for the offscreen render target categories.
- The SoftwareLayer properties are specified in the Candera::SoftwareLayerProperties class (see Candera::SoftwareLayer::GetSoftwareLayerProperties)
- and the pixel format information is kept in the Candera::GlFrameBufferProperties class (see Candera::SoftwareLayer::GetFrameBufferProperties()).

EGL Context Handling

Description

EGL is an interface between rendering APIs like OpenGL ES and the underlying native platform window system. Rendering APIs like OpenGL (ES) and OpenVG use EGL contexts to manage their states and resources.

EGL Concepts

EGL Context: WindowSurface

An EGL context typically stores render states, shaders, textures, vertex buffers and framebuffer resources. In Candera each WindowSurface (onscreen framebuffer) creates an EGL context when uploaded. Thus, each WindowSurface owns its separate context to store related render states, like for instance depth buffer, and blend mode settings.

EGL Context Sharing: ContextResourcePool

An EGL context can also be shared to store resources like textures, vertex buffer objects, and shaders. This feature avoids redundant video memory uploads of shared resources to multiple EGL contexts and reduces memory footprint. Candera provides context sharing functionality with class ContextResourcePool. Each display is associated to exactly one context resource pool. One context resource pool can be shared accross different displays, if supported by platform. In case context sharing is not supported, for instance as displays are driven by separate display controllers, Candera allows to upload resources of a single scene graph to multiple EGL contexts.

Framebuffer Objects using ContextResourcePool

Framebuffer objects (FBOs) are defined in OpenGL ES API and do, different to WindowSurfaces, not own a dedicated EGL context. Instead, FBOs require a context provided from an EGL surface to store related data in video memory. In Candera a WindowSurface is such a context provider, which registers its EGL context to ContextResourcePool to be shared. FBOs

Uploading To Multiple Contexts

Overview

A device object can be uploaded to multiple context resource pools but there is no association from a device object to its context resource pools, before the uploading. Further, AssetLoader could upload a device object to a context resource pool activated accidentally. In order to prevent such

situations, Candera offers a dedicated interface meant to give more control in the process of uploading/unloading device objects in multiple contexts.

Multiple Contexts Uploading

The uploading to more then one context at a time can be accomplished by the means of the Node::UploadAll() method. There is also a method, Node::UnloadAll(), which does the reverse operation, the unloading from video memory.

These methods take two parameters, a scope mask and a multi context flag. The *multiContext* flag controls if the operation is performed strictly on the active context or on all existing context resource pools belonging to the cameras which are in the scope of the upload/unload operations. The *scopeMask* parameter is useful to filter the nodes which are to be uploaded/unloaded from scenes containing multiple nodes. The algorithm of filtering using the scope mask works as follows:

- First it creates a list of cameras which are in the scope of the upload/unload operation.
- Then, for each of these cameras, it will iterate over the nodes "seen" by the camera and it will upload/unload only the nodes (and their descendants) which are in the scope of the camera intersected with the scope of the upload/unload operation. The context resource pool targeted by this operation is the one belonging to the camera.

For a better understanding of the feature, please have a look on the example presented on the next item.

Code Example

```
Candera::ScopeMask m_scopeMask;
m_scopeMask.SetAllScopesEnabled(false);
m_scopeMask.SetScopeEnabled(2, true);

m_group->UploadAll(m_scopeMask, true);
//...
m_group->UnloadAll(m_scopeMask, true);
```

Uploading To Multiple Contexts Example

Introduction

The example below tries to offer you a better understanding of the way the method Candera::Node::UploadAll() / Candera::Node::UnloadAll() works. In order to do that, we chose a complex configuration in which the nodes to be uploaded, the cameras and the uploading method itself, are set with different values for the scope masks.

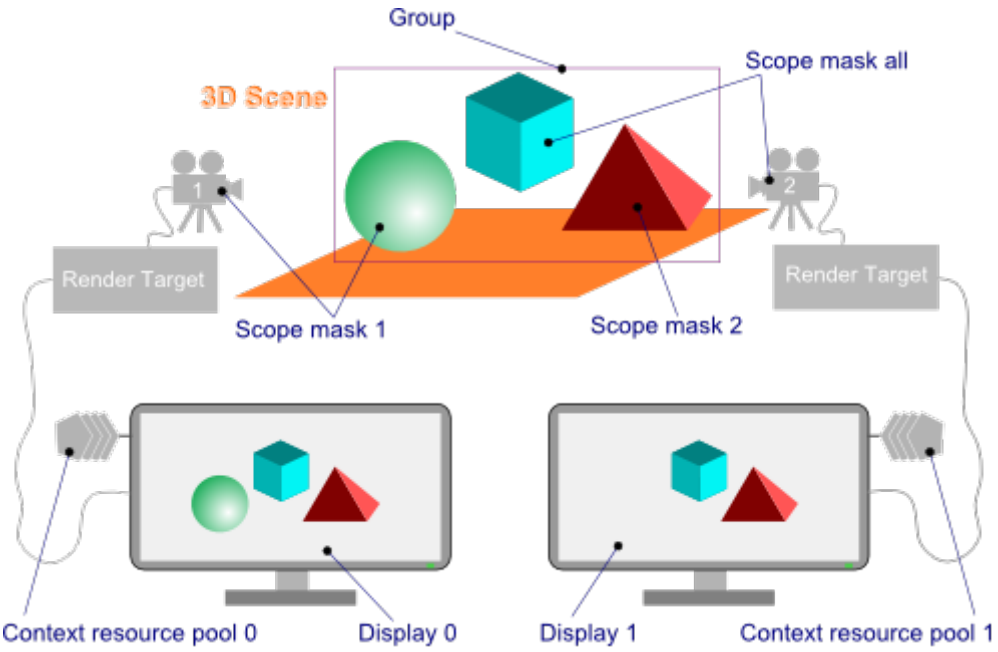
Scene Setup

The setup consists in a 3D scene which contains three nodes: a sphere, a cube and a pyramid, all three nodes are children of a Candera::Group. The 3D content is rendered by two cameras on two displays.

We assume that the context resource pools associated to the displays are not shared, so each display will have its own context resource pool. Each of these nodes and cameras have assigned a distinct Candera::ScopeMask as follows:

Object	Scope Mask
Sphere	1
Pyramid	2
Cube	All
Camera 1	1
Camera 2	All

The scene setup is shown also in the image below:



The mask scopes are created as follows:

```
Candera::ScopeMask m_scopeMask1;
m_scopeMask1.SetAllScopesEnabled(false);
m_scopeMask1.SetScopeEnabled(1, true);

Candera::ScopeMask m_scopeMask2;
m_scopeMask2.SetAllScopesEnabled(false);
m_scopeMask2.SetScopeEnabled(2, true);
```

```
Candera::ScopeMask m_scopeMaskAll;
m_scopeMaskAll.SetAllScopesEnabled(true);
```

Then they are set as scope masks for the nodes:

```
m_meshSphere->SetScopeMask(m_scopeMask1);
m_meshPyramid->SetScopeMask(m_scopeMask2);
m_meshCube->SetScopeMask(m_scopeMaskAll);
m_camera1->SetScopeMask(m_scopeMask1);
m_camera2->SetScopeMask(m_scopeMask2);
```

> For the convenience, we will assume that the last context activated by the application is Context Resource Pool 1.

Upload Scenarios

The table below shows in yellow the input parameters for the UploadAll method and in blue whether the nodes are uploaded or not and in which context.

For example, the first row in the table data says that the UploadAll method was called with the parameters:

```
m_group->UploadAll(/*scopeMask=*/m_scopeMask1, /*multiContext=*/true);
```

Fifth row correspond to the following call:

```
m_group->UploadAll(/*scopeMask=*/m_scopeMask2, /*multiContext=*/false);
```

And so on.

Index	MultiContext flag	Upload scope mask			Uploaded in CRP 0			Uploaded in CRP 1		
		1	2	All	Cube	Sphere	Pyramid	Cube	Sphere	Pyramid
1	✓	✓	✗	✗	✓	✓	✗	✓	✓	✗
2	✓	✗	✓	✗	✗	✗	✗	✓	✗	✓

3	✓	✗	✗	✓	✓	✓	✗	✓	✓	✓
4	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗
5	✗	✗	✓	✗	✗	✗	✗	✓	✗	✓
6	✗	✗	✗	✓	✗	✗	✗	✓	✓	✓

Result explanation:

1) Because multi context flag is enabled, the application will try to upload device objects in both context resource pools. Both cameras are in the scope of the upload (scope 1). Camera 1, corresponding to the Context Resource Pool 0 has the scope mask 1 so the application will upload the Cube (scope mask all) and the Sphere (scope mask 1) but not the pyramid (scope mask 2). In the Context Resource Pool 1, corresponding to the Camera 2 (scope mask all) the application will upload Cube (scope mask all) and the Sphere (scope mask 1). The pyramid will not be uploaded because it is not in the scope of upload intersected with scope of the camera (1 & "all" = 1).

2) Because multi context flag is enabled, the application will try to upload device objects in both context resource pools but only Camera 2 is in the scope of the upload (scope mask 2). Consequently, application will not upload anything in the Context Resource Pool 0 corresponding to the Camera 1. In the Context Resource Pool 1, corresponding to the Camera 2 (scope mask all), the application will upload the Cube (scope mask all) and the Pyramid (scope mask 2). The Sphere (scope mask 1) will not be uploaded because it is not in the scope of upload intersected with scope of the camera (2 & "all" = 2).

3) Because multi context flag is enabled, the application will try to upload device objects in both context resource pools. Both cameras are in the scope of the upload (scope maks all). Camera 1, corresponding to the Context Resource Pool 0 has the scope mask 1 so the application will upload the Cube (scope mask all) and the Sphere (scope mask 1). The pyramid will not be uploaded because it is not in the scope of upload intersected with scope of the camera ("all" & 1 = 1). In the Context Resource Pool 1, corresponding to the Camera 2, all three nodes will be uploaded.

4) The multi context flag is disabled, so the upload will be performed only in the last activated context (Context Resource Pool 1). Consequently, the application will not upload anything in the Context Resource Pool 0. Camera 2 (scope mask all) is in the scope of the upload (scope mask 1) so it will be taken into account for the upload. In the Context Resource Pool 1, corresponding to the Camera 2 (scope mask all) the application will upload Cube (scope mask all) and the Sphere (scope mask 1). The pyramid will not be uploaded because it is not in the scope of upload intersected with scope of the camera (1 & "all" = 1).

5) The multi context flag is disabled, so the upload will be performed only in the last activated context (Context Resource Pool 1). Consequently, the application will not upload anything in the Context Resource Pool 0. Camera 2 (scope mask all) is in the scope of the upload (scope mask 2) so it will be taken into account for the upload. In the Context Resource Pool 1, corresponding to the Camera 2 (scope mask all), the application will upload the Cube (scope mask all) and the Pyramid (scope mask 2). The Sphere (scope mask 1) will not be uploaded because it is not in the scope of upload intersected with scope of the camera (2 & "all" = 2).

6) The multi context flag is disabled, so the upload will be performed only in the last activated context (Context Resource Pool 1). Consequently, the application will not upload anything in the Context Resource Pool 0. Camera 2 (scope mask all) is in the scope of the upload (scope mask 1) so it will be taken into account for the upload. In the Context Resource Pool 1, corresponding to the Camera 2, all three nodes will be uploaded because their scope are in the scope of upload intersected with scope of the camera ("all" & "all" = "all") .

Multiple Render Targets

Description

This section explains how to handle Multiple Render Targets by means of a small ping pong like tutorial solution.

Example Solution

- RenderTargetsCamerasSolution in folder *cgi_studio_player/content/Tutorials/07_RenderTargetsCameras*

Handling Multiple Render Targets

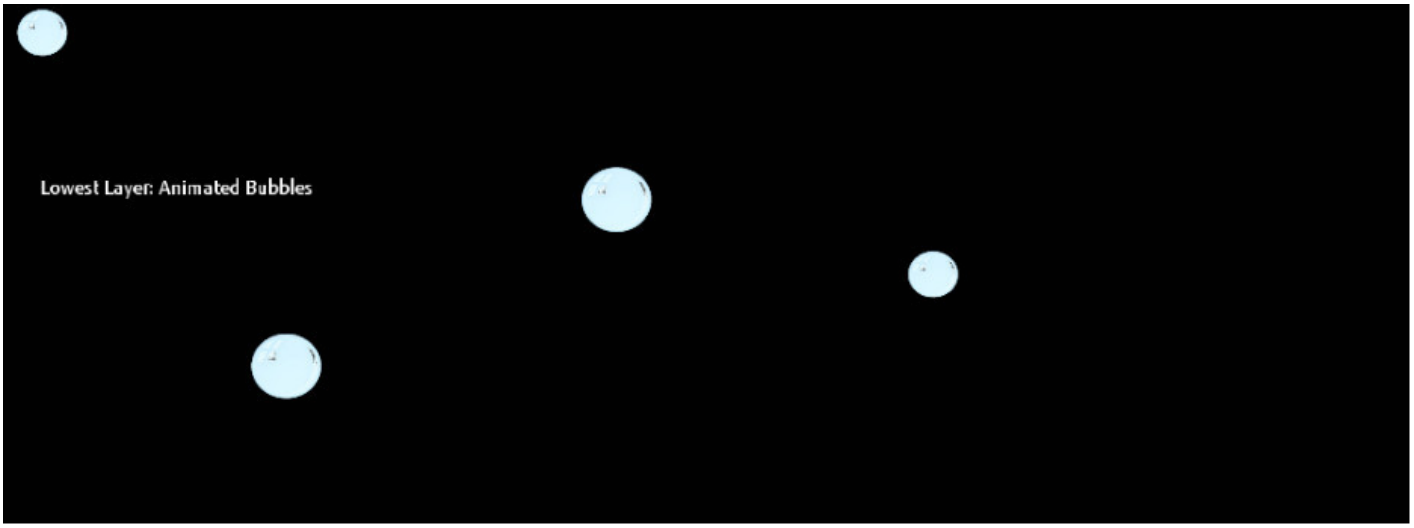
The example solution in *cgi_studio_player/content/Tutorials/07_RenderTargetsCameras* consists of the following:

- a main scene rendered to Candera::iMX6WindowSurface render target
- three different scenes rendered to three different Candera::SoftwareLayer render targets.

The three Candera::SoftwareLayer render targets are linked to the Candera::iMX6WindowSurface render target through their **Compostion Camera** property and will be blended against each other.

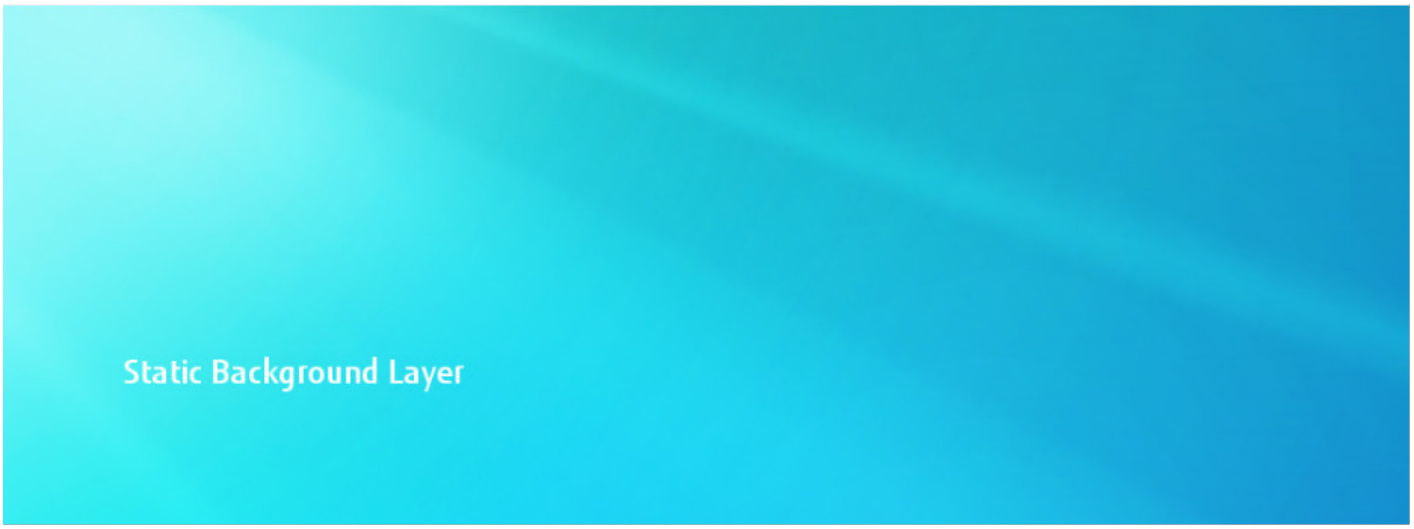
Layer 1: Animated 2D Background

The lowermost layer is a collection of bubbles which are animated to bubble endlessly from the bottom of the 2D screen to the top. It is the lowest layer, because the dreary, transparent bubbles shall be colored by blending with a more colorful layer.



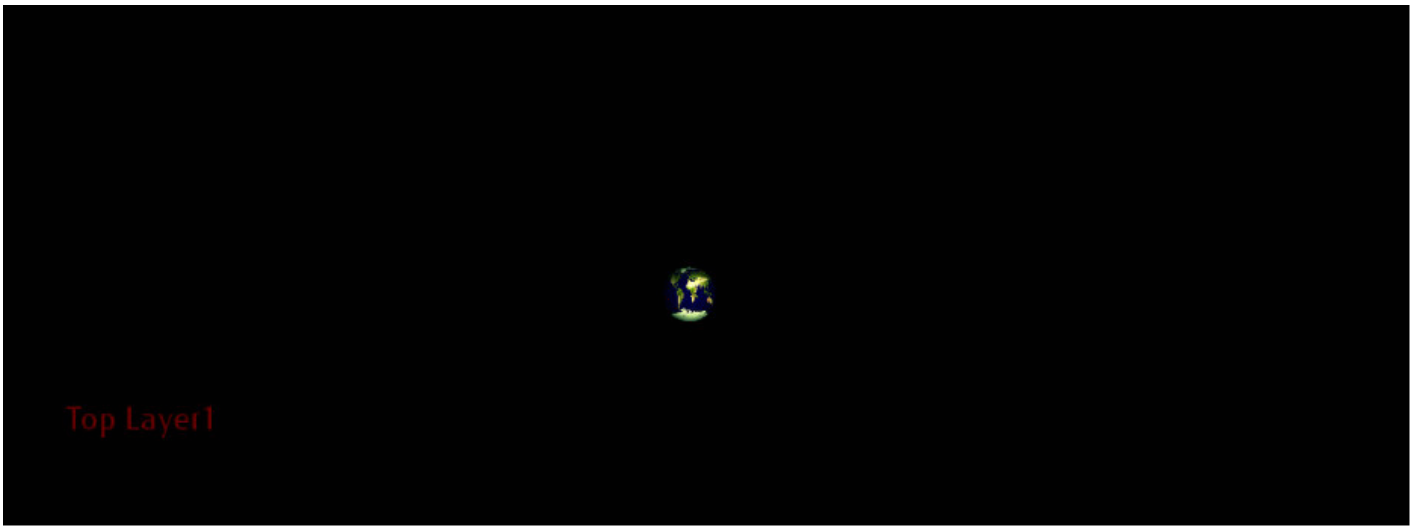
Layer 2: Static 2D Background

This is the layer which will give the animated bubbles some colors.



Layer 3: Dynamic 3D Top Layer

Finally, the top layer shows a 3D scene with a sphere.



The sphere can be linked with the "BouncingNodeWidget". Once this widget is enabled in the Player, it will send the sphere into several directions like in a Ping Pong game. This is the reason, why the camera of this scene needs an OrthographicProjection: The widget needs to match the sphere position on the x and y axis with the boundaries of the 2D projection plane.

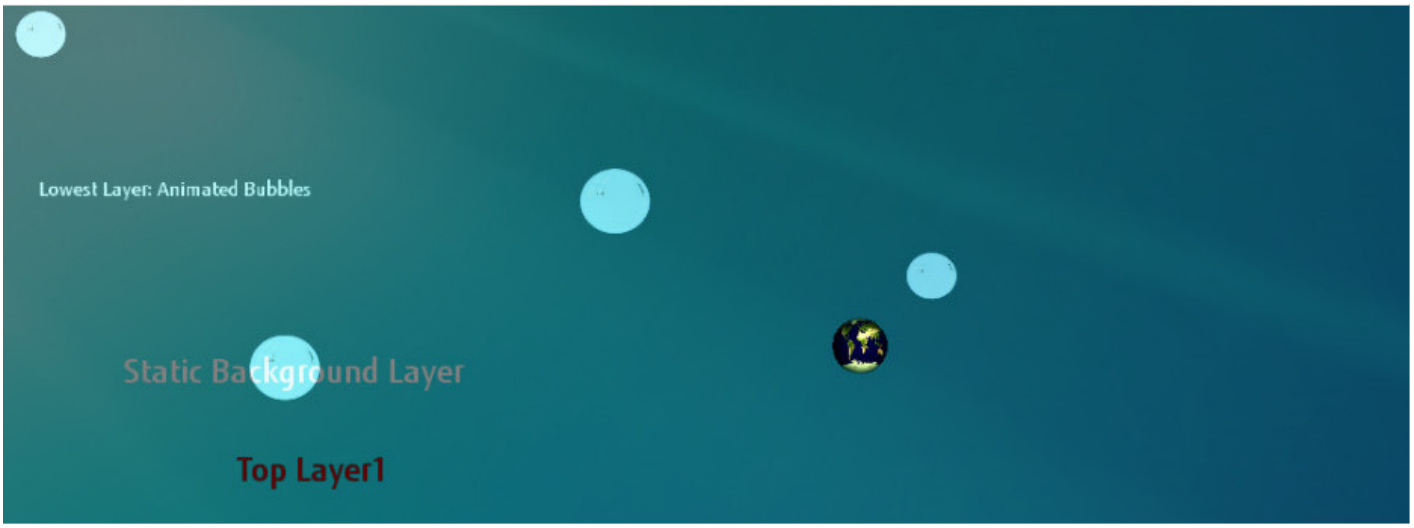
Use the velocity widget property to play with different velocities.

Layer Order and Layer Blending

Layer Order from Lowest to Top

The correct layer order must be specified on the property "Layer" on the render targets, the bigger value for the top layer:

- Lowest Layer: Layer-ID 0
- Medium Layer: Layer-ID 1
- Top Layer: Layer-ID 2



Export Asset and Load with the Player

- Export the example solution to a simulation asset
- When the asset is loaded in the Player, take care to first activate the main scene (Scene3DOwner from solution) and then the other scenes.

Render Target Manipulation

Accessing Render Target Properties

The BouncingNodeWidget uses a property to specify the name of a layer. Width and height of this layer is used by the widget to let the node bounce within these boundaries.

Refer to the following code example to access width and height of this layer:

```
        if (m_layer != 0){
            RenderTarget3D* rt3d = m_layer->ToRenderTarget3D();
            if (rt3d != 0){
                m_layerWidth = rt3d->GetWidth();
                m_layerHeight = rt3d->GetHeight();
            }
        }
```

Accessing Render Target Properties by MetaInfo

Same properties could be accessed by meta information as follows:

```
    const MetaInfo::GraphicDeviceUnitMetaInfo *meta = DevicePackageDescriptor::GetMetaInformatic
    if (meta){
        if (meta->LookupItem("Width") != 0){
            meta->LookupItem("Width")->Get(gdu,m_width,sizeof(m_width));
        }
        if (meta->LookupItem("Height") != 0){
            meta->LookupItem("Height")->Get(gdu,m_height,sizeof(m_height));
        }
        if (meta->LookupItem("BlendMode") != 0){
            meta->LookupItem("BlendMode")->Get(gdu, m_blendMode,sizeof(m_blendMode));
        }
        // ...
    }
```

Accessing Software Layer Properties

The code snippet below shows how to set the software layer properties for an iMX6 device.

```
    Candera::GraphicDeviceUnit* m_gdu = Candera::DevicePackageInterface::CreateGraphicDeviceUnit
    static_cast<Candera::SoftwareLayer*>(m_gdu)->GetSoftwareLayerProperties().CompositionCamera(
    //...
```

Texture Render Targets

Description

This section explains how to handle Texture Render Targets in 2D and 3D.

Example Solutions

- TextureRenderTargetsSolution in folder *cgi_studio_player/content/Tutorials/07_RenderTargetsCameras*
- MixedRenderTargetsSolution in folder *cgi_studio_player/content/Tutorials/07_RenderTargetsCameras*

Define Texture Render Targets

To illustrate the usage of an OffScreenTarget2D, open the solution cgi_studio_player/content/Tutorial/07_RenderTargetCameras/TextureRenderTargetsSolution.

Offscreen Scene Content

First of all, "offscreen" scenes must be defined, which shall later be rendered into textures:

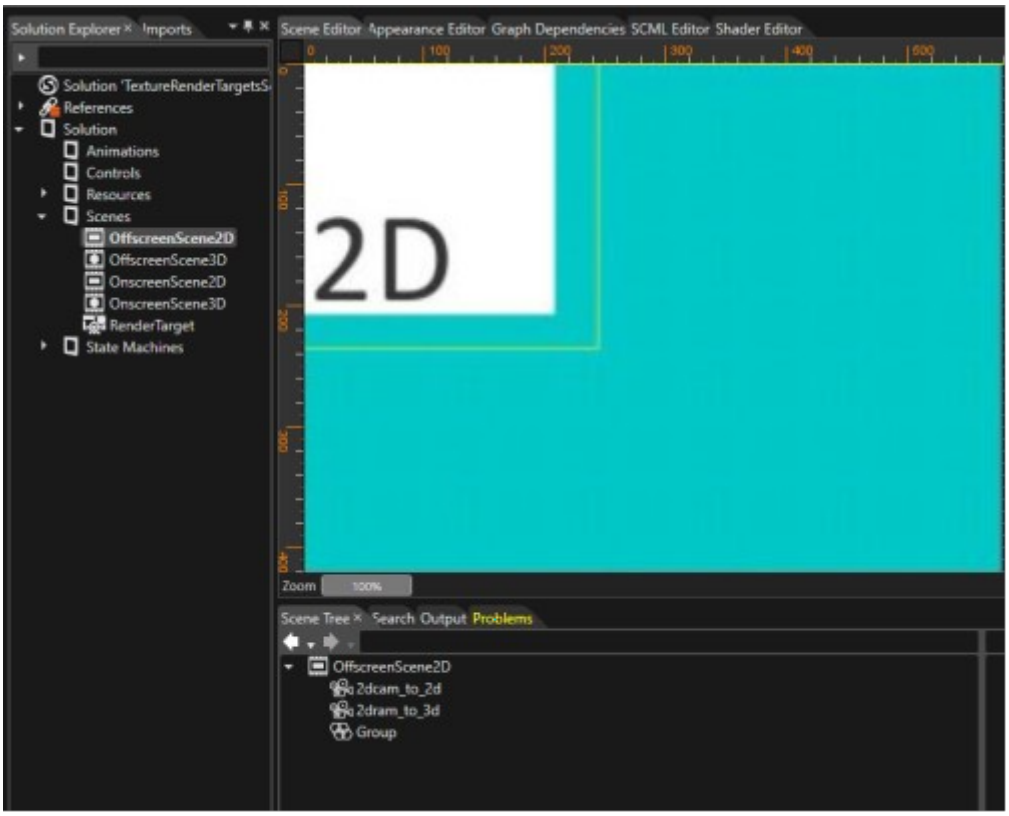
- **OffscreenScene2D:** This 2D scene contains a white box with the text "2D" in it.
- **OffscreenScene3D:** This 3D scene contains a Text3DWidget with text "3D" and a light.

Offscreen Scene Cameras

Next, cameras must be defined for rendering the offscreen scenes into textures.

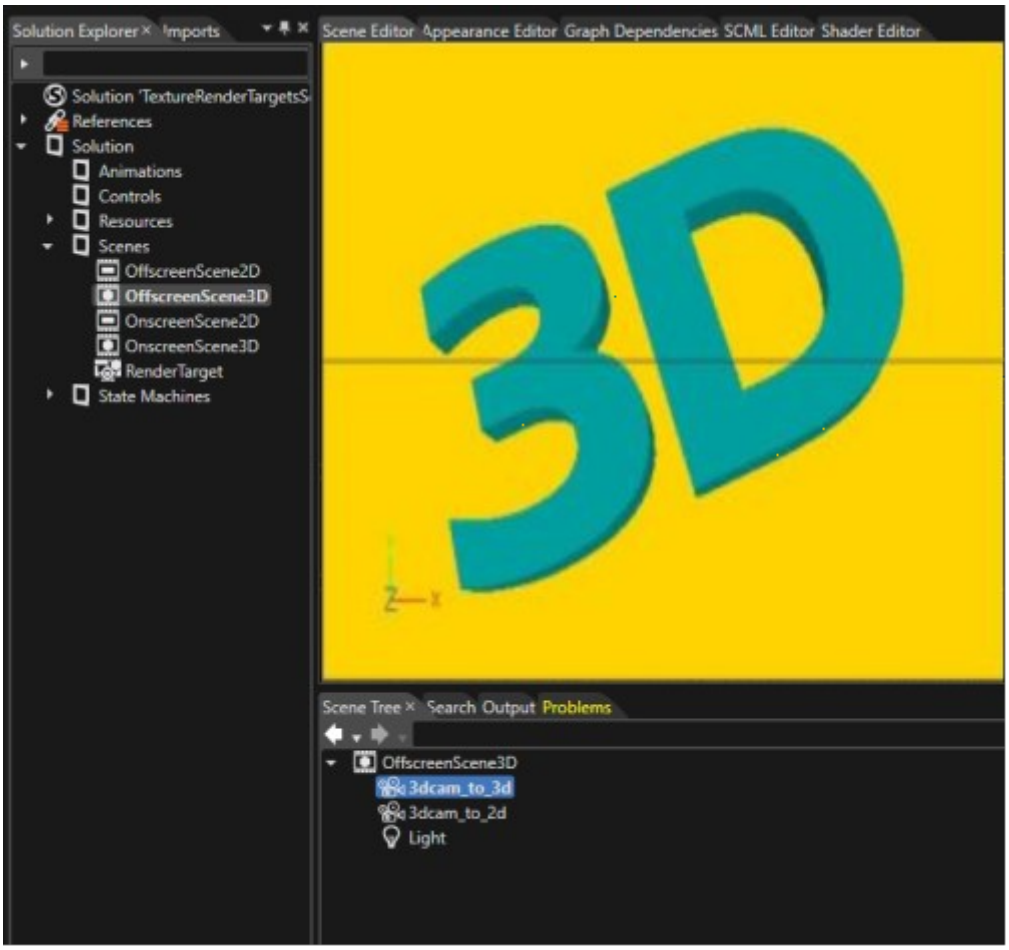
In case of the **OffscreenScene2D**, two different cameras are defined:

- **2dcam_to_2d**: This camera will be linked to a **OffscreenRenderTarget2D**, for rendering into a 2D texture.
- **2dcam_to_3d**: This camera will be linked to a **OffscreenRenderTarget2Dto3D**, for rendering into a 3D texture.



Similarly, the **OffscreenScene3D** defines following two cameras:

- **3dcam_to_3d**: This camera will be linked to a **OffscreenRenderTarget3D**, for rendering into a 3D texture.
- **3dcam_to_2d**: This camera will be linked to a **OffscreenRenderTarget3Dto2D**, for rendering into a 2D texture.



Define Offscreen Render Targets

Finally, all the cameras from Offscreen scenes must be linked to the related offscreen render targets, for rendering the content of offscreen scenes:

- **OffscreenTarget2D:** 2D render target for the text "2D"
- **OffscreenTarget2Dto3D:** 3D render target for the text "2D"
- **OffscreenTarget3D:** 3D render target for the text "3D"
- **OffscreenTarget3Dto2D:** 2D render target for the text "3D"

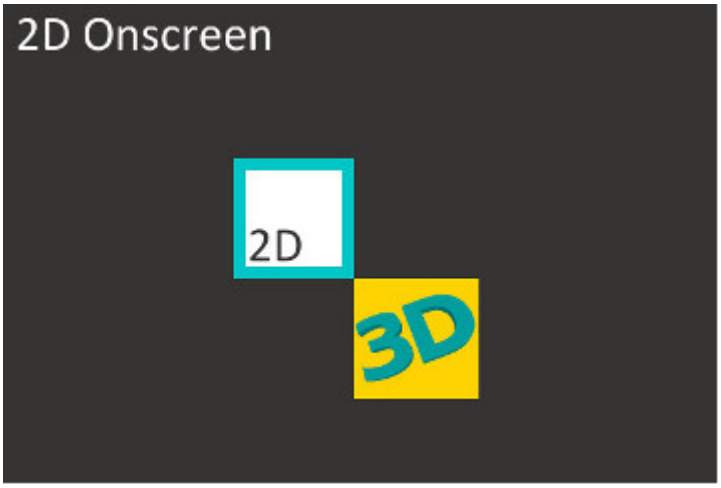
Use Texture Render Targets

Use Texture Render Targets in a 2D Scene

The 2D scene "OnscreenScene2D" is rendered on a DisplayRenderTarget2D and shall include the render outputs from both the 2D and 3D offscreen scenes.

For this, bitmap nodes are defined in this scene which use the following offscreen render targets as textures:

- **OffscreenTarget2D**
- **OffscreenTarget3Dto2D**

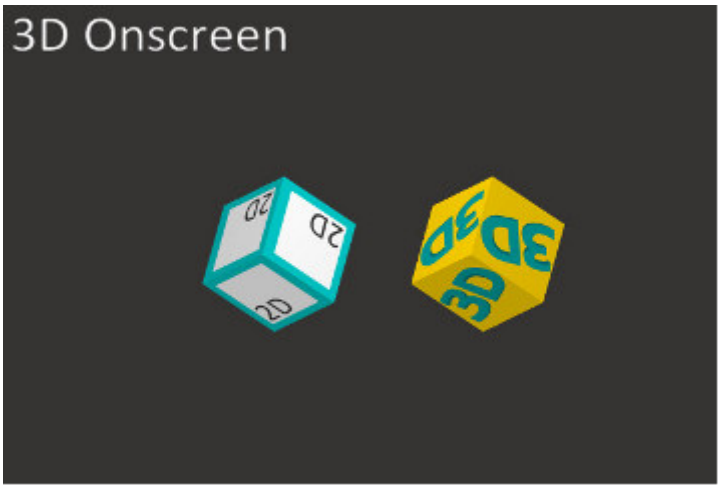


Use Texture Render Targets in a 3D Scene

The 3D scene "OnscreenScene3D" is rendered on a DisplayRenderTarget3D and shall include the render outputs from both the 2D and 3D offscreen scenes.

For this, two cubes are defined in this scene which use the following offscreen render targets as textures:

- **OffscreenTarget3D**
- **OffscreenTarget2Dto3D**



Composition Camera Property Set

The **Composition Camera** property of the DisplayRenderTarget3D and DisplayRenderTarget2D render targets must be set to the camera linked with RenderTargetOwner. These is mandatory for rendering the composed contend of the two display render targets.

> The Layer property must be correctly set for Candera::SoftwareLayer render targets (DisplayRenderTarget3D and DisplayRenderTarget2D). The layers must have different values, with the lowest value for the lowest layer.

Example Solution

Another example of using Texture Render targets can be seen in the tutorial solution *cgi_studio_player/content/Tutorials/07_RenderTargetsCameras/MixedRenderTargetsSolution*.

OffscreenTarget2Dto3D, OffscreenTarget3Dto2D and Mixed2D3DOffscreenTarget render targets are used as texture for 3D meshes or as Image for BitmapBrush effects. The Mixed2D3DDisplayTarget is linked to the display and can be used for 2D and/or 3D content. In the tutorial solution *MixedRenderTargetsSolution* there are two cameras (2D and 3D) attached to the Mixed2D3DDisplayTarget. The resulted image on display shows the content of the both scenes overlapped.

> **Rules to configure multiple cameras attached to a render target**
> - Camera 2D: Clear Color Buffer = false; Enable Swapping = true
> - Camera 3D: Clear Color Buffer = true; Enable Swapping = false

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

[1] www.shutterstock.com/pic-12442312/stock-vector-a-business-man-runs-amp-power-walks-to-success-in-animation-sequence-frame-loops-with.html

Camera Groups

Description

This section explains how to use Camera Groups to handle cameras uniformly across multiple scenes.

Example Solution

- CameraGroupSolution example in folder *cgi_studio_player/content/Tutorials/07_RenderTargetsCameras*

Camera Group - Interface

Candera::CameraGroup is a container that groups an arbitrary number of 2D and/or 3D cameras from different scenes. This feature is especially useful for complex applications because it offers a simpler way to control which cameras are rendering at a certain moment.

The snippet below shows how to add cameras to a Candera::CameraGroup object:

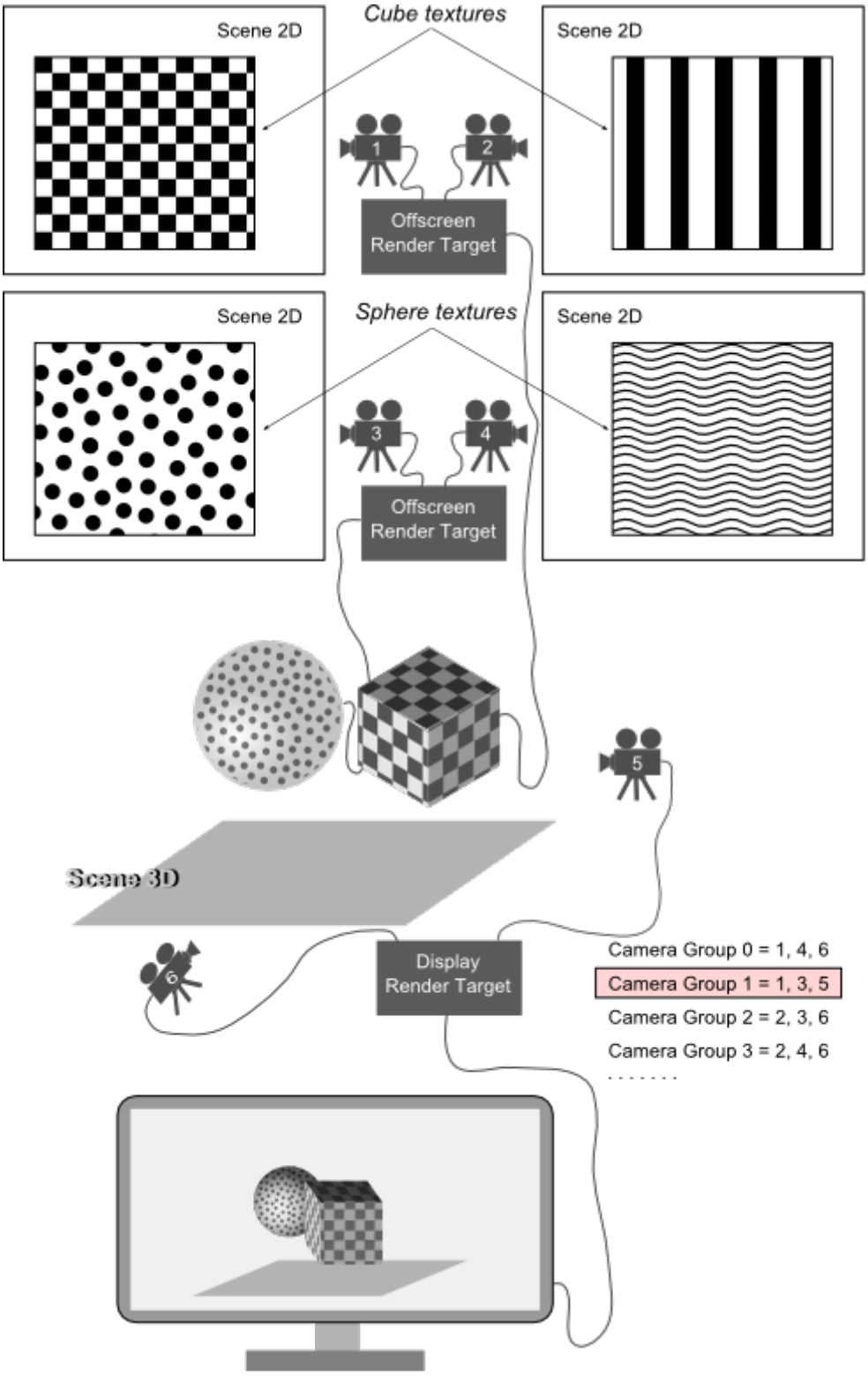
```
    m_cameraGroup[0].AddCamera2D(m_cameraCubeTex1);
    m_cameraGroup[0].AddCamera2D(m_cameraSphereTex1);
    m_cameraGroup[0].AddCamera(m_cameraFront);

    m_cameraGroup[1].AddCamera2D(m_cameraCubeTex2);
    m_cameraGroup[1].AddCamera2D(m_cameraSphereTex1);
    m_cameraGroup[1].AddCamera(m_cameraFront);

    m_cameraGroup[2].AddCamera2D(m_cameraCubeTex2);
    m_cameraGroup[2].AddCamera2D(m_cameraSphereTex2);
    m_cameraGroup[2].AddCamera(m_cameraFront);

    m_cameraGroup[3].AddCamera2D(m_cameraCubeTex2);
    m_cameraGroup[3].AddCamera2D(m_cameraSphereTex2);
    m_cameraGroup[3].AddCamera(m_cameraSide);

    m_cameraGroup[4].AddCamera2D(m_cameraCubeTex1);
    m_cameraGroup[4].AddCamera2D(m_cameraSphereTex2);
    m_cameraGroup[4].AddCamera(m_cameraSide);

    m_cameraGroup[5].AddCamera2D(m_cameraCubeTex1);
    m_cameraGroup[5].AddCamera2D(m_cameraSphereTex1);
    m_cameraGroup[5].AddCamera(m_cameraSide);
```

Then, any group can be enabled for rendering as follows:

```
    m_cameraGroup[idx].SetRenderingEnabled(true);
    if (idx < 3){
        m_cameraFront->SetSwapEnabled(true);
        m_cameraSide->SetSwapEnabled(false);
    }
    else{
        m_cameraFront->SetSwapEnabled(false);
        m_cameraSide->SetSwapEnabled(true);
    }
```

Camera Groups - Example

The diagram below presents a brief example of how to dynamically set 3D nodes textures by enabling the rendering of different camera groups.



The texture of each of the two nodes from the 3D scene is an offscreen render target linked to a pair of cameras 2D. Each of the 2D cameras is "seeing" a pattern which is "projected" on the node surface when the camera is enabled for rendering. The cameras are added in CameraGroups, each camera group contains three cameras: a camera 3D and two camera 2D corresponding to the cube and sphere textures. By enabling/disabling the rendering of different camera groups the textures on the nodes will change accordingly. For best results enable for rendering only one camera group at a time.

> A camera can belong to multiple groups, but its state is unique, meaning that enabling the camera in one group will enable it in all groups and also in the Render Target panel where it is associated to a render target. If a group is disabled or enabled, all the cameras inside it will be disabled or enabled. If one or many from those cameras are members of another group, the second group will be enabled or disabled too.

See also:

- Camera Group in SceneComposer User Manual

Example Solution

- CameraGroupSolution example in folder *cgi_studio_player/content/Tutorials/07_RenderTargetsCameras* illustrates the scenario presented above. In order to enable/disable camera groups in SceneComposer, select the Render Targets panel and press on the third button showing the tooltip "Go to Camera Group Explorer view".

Proxy Texture Image

Description

This section describes how to use the proxy texture images and how to create custom image sources.

Texture Images

Candera offers out of the box usage of 2D and CubeMap textures in 3D scenes, which is sufficient in many cases. These two texture types are directly represented through the classes

- Candera::BitmapTextureImage and
- Candera::CubeMapTextureImage.

However, these two derivations of Candera::TextureImage don't represent all use cases that can be realized with textures.

ProxyTextureImage

For this reason a third derivation, namely, Candera::ProxyTextureImage exists, that simply forwards an external texture handle to the texture activation procedures, in order to display other input than the previously mentioned image data. Detailed description of usage is given in chapter Proxy Texture Images.

One use case that can be realized with Candera::ProxyTextureImage is the usage of off-screen render targets in order to use arbitrary rendered images of 3D scenes as texture image. Refer to the first example part of this tutorial:

- Example: Offscreen Render Targets

Other use cases, such as video textures can only be realized using (often proprietary) OpenGL ES 2.0 extensions. How to create an ImageSource3D that can be used for these use cases is the second part of this tutorial. Refer to:

- Example: External Image Source

Proxy Texture Images

In Candera Candera::TextureImage provides an interface to upload and activate a textures VRAM handle. This interface is further used by Candera::Texture to separate the common settings of textures and the specialized texture types. This chapter focuses on the use of Candera::ProxyTextureImage, for details on texturing itself, please refer to the Appearance Tutorial on Textures.

Knowledge of the following Candera::TextureImage functions is needed to understand Candera::ProxyTextureImage.

```
virtual Handle GetVideoMemoryHandle() const = 0;
```

exposes a textures VRAM handle to all clients that use Candera::TextureImage. This is needed in order to enable texture render states and to activate the texture.

```
virtual bool IsMipMappingEnabled() const = 0;
```

returns if mip mapping is enabled on a texture or not.

These functions are directly implemented and coupled to fixed functionality in the derivations Candera::BitmapTextureImage and Candera::CubeMapTextureImage.

Candera::ProxyTextureImage also implements the Candera::TextureImage interface, but in contrast to the other implementations it takes Candera::ImageSource3D objects, and simply forwards their VRAM handles through the GetVideoMemoryHandle function. Setting of the ImageSource can only be done at creation time, use the ProxyTextureImage's factory function for doing this.

```
static MemoryManagement::SharedPointer<ProxyTextureImage> Create(ImageSource3D* imageSource);
```

ImageSource3D Interface

As explained in the previous section, Candera::ProxyTextureImage takes an Candera::ImageSource3D and forwards its VRAM handle through the Candera::TextureImage interface. This section explains what the Candera::ImageSource3D interface is, and how it's used.

A Candera::ImageSource3D object in Candera is basically an interface that provides a handle to an OpenGL ES 2.0 texture in VRAM. It is derived from Candera::ImageSource, which inherits from Candera::Surface which itself inherits from Candera::Synchronizable.

Therefore the following interface has to be implemented:

- Candera::Synchronizable Methods

  ```
  virtual void Sync() = 0;
  ```

  If some kind of synchronization needs to be done, implement it inside this method. It will be called every time the Candera::ProxyTextureImage, where this ImageSource3D is associated to, gets activated. Prerequisite is that Candera::ProxyTextureImage::IsSyncEnabled is set to true.

  ```
  virtual void WaitSync() = 0;
  ```

  Blocks the current thread until the object is in synchronized state. In the context of Candera::ProxyTextureImage this is never called, but has to be implemented at least empty.

- Candera::Surface Methods

  ```
  virtual Int GetHeight() const = 0;
  ```

  ```
  virtual Int GetWidth() const = 0;
  ```

  These functions have to return the width and the height of the texture in texels.

- Candera::ImageSource Methods

No special interface to implement.

- Candera::ImageSource3D Methods

  ```
  virtual Handle GetVideoMemoryHandle() const = 0;
  ```

  Exposes the video memory handle of the Candera::ImageSource3D implementation. This is the most important function to create ImageSources as this is the handle which gets forwarded from Candera::ProxyTextureImage::GetVideoMemoryHandle. This handle is further used for setting render states and activation of the texture. Note: This handle has also to be uploaded and eventually updated, which is not done by Candera itself, but has to be done by the application. Imagine e.g. a video texture using an appropriate extension, the handle has to be uploaded before rendering. The content of the VRAM buffer then has to be updated before the Candera::Renderer::RenderCamera or Candera::Renderer::RenderAllCameras is called.

  ```
  virtual bool IsMipMappingEnabled() const = 0;
  ```

  Returns if mip mapping is enabled or not. Set it to your needs or make it configurable.

This interface is used for multiple purposes:

- Candera uses it for multiple device objects, such as off-screen render targets. With Candera::ProxyTextureImage this is used to realize rendering scenes into textures.
- Existing Candera::TextureImage derivations like Candera::BitmapTextureImage and Candera::CubeMapTextureImage offer a function ToImageSource3D and can therefore also be used with Candera::ProxyTextureImage.
- Own ImageSources that use e.g. an OpenGL ES extension for texturing have to implement the ImageSource3D interface and can also be used with Candera::ProxyTextureImage.

Example: Offscreen Render Targets

Offscreen Render Targets

The following source code example demonstrates the additional steps that need to be done in order to use offscreen render targets with Candera::ProxyTextureImage.

> For specific devices the prefix "Device" is used in the beginning of class identifiers. Simply exchange it with the according device name.

First, after display creation, create and configure an off-screen render target:

```
DeviceFrameBufferObject* CreateFrameBufferObject() {
    Int count = 0;
    const GraphicDeviceUnitTypeHandle* handle = DevicePackageDescriptor::GetUnitTypes(DevicePack
    if (handle == 0 || count < 1){
        return 0; //Handle creation error.
    }
    GraphicDeviceUnit* gdu = DevicePackageInterface::CreateGraphicDeviceUnit(handle[0]);
    if (gdu != 0) {
        gdu->SetDisplay(displayId);
    }

    return (DeviceFrameBufferObject*)gdu;
}

GraphicDeviceUnit* m_gdu;
DeviceFrameBufferObject* m_fbo;
```

```
void Init() {
    ...
    //Display creation
    m_gdu = ...;
    ...
    m_gdu->Upload();

    //Create and configure offscreen render target.
    m_fbo = CreateFrameBufferObject();
    m_fbo->SetWidth(m_gdu->ToRenderTarget3D()->GetWidth());   //Offscreen RT now has same dimens
    m_fbo->SetHeight(m_gdu->ToRenderTarget3D()->GetHeight()); //Of course this can vary.
    m_fbo->SetColorFormat(RgbaUnpackedColorFormat);
    m_fbo->SetDepthFormat(Depth32Format);
    m_fbo->Upload();
    ...
    InitCamera();
    InitScene();
    m_scene->UploadAll();
}
```

Further specify a camera which renders the offscreen render target. This is basically the same as normal camera creation, except that the fbo is set as render target.

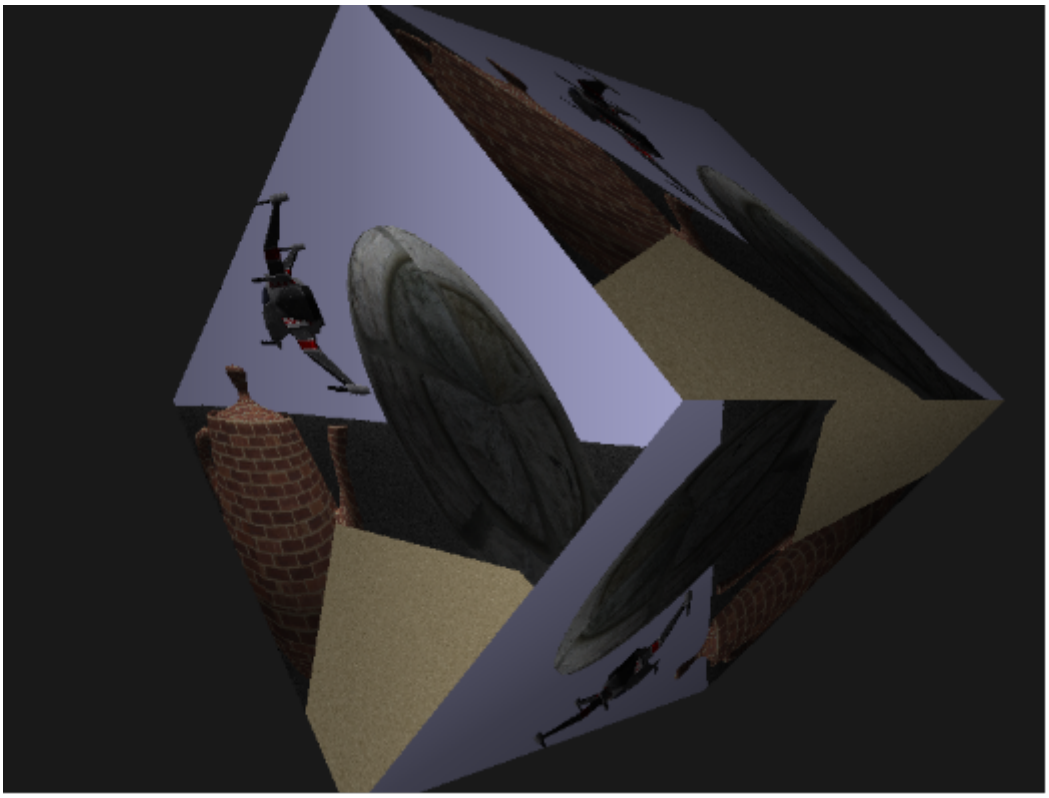
```
void InitCamera() {
    //Initialize main camera
    m_camera = Camera::Create();
    ...

    //Initialize offscreen camera
    m_offscreenCam = Camera::Create();
    m_offscreenCam->SetName("OffScreen Camera");
    m_offscreenCam->SetViewport(Candera::Rectangle(0.0f, 0.0f, 1.0f, 1.0f));
    //Here the offscreen RT is used as camera render target.
    m_offscreenCam->SetRenderTarget(m_fbo->ToRenderTarget3D());

    const Int vpWidth = 1600;//m_width;
    const Int vpHeight = 600;//m_height;

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

    m_offscreenCam->SetProjection(perspectiveProjection);
    m_offscreenCam->SetPosition(Vector3(0.0f, 0.0f, 0.0f));
    m_offscreenCam->SetUpVector(Vector3(0.0f, 1.0f, 0.0f));
    m_offscreenCam->LookAtWorldPoint(Vector3(0.0f, 0.0f, -1.0f));
```

```
    m_clearMode.SetClearColor(Color(0.8f,0.8f,1.0f,1.0f));
    m_clearMode.SetColorClearEnabled(true);
    m_clearMode.SetDepthClearEnabled(true);
    m_clearMode.SetClearDepth(1.0f);
    m_offscreenCam->SetClearMode(m_clearMode);
    m_offscreenCam->SetClearModeEnabled(true);
    m_offscreenCam->SetSwapEnabled(true);

    m_scene->AddChild(m_offscreenCam); //You can off course also render different scenes
                                       //your offscreen camera.
}
```

Finally, set the off-screen render target as image source of the proxy texture.

```
void InitScene() {
    //Init your scenes here
    ...
    //The texture the offscreen RT is rendered into.
    SharedPointer<Texture> proxyTex = Texture::Create();
    //here the fbo is set as the textures image source!
    SharedPointer<ProxyTextureImage> proxyTexImg = ProxyTextureImage::Create
(m_fbo->ToImageSource3D());
    //Use the ProxyTextureImage as the textures TextureImage.
    proxyTex->SetTextureImage(proxyTexImg);
    //Set up the texture to fit your needs.
    proxyTex->SetMipMapFilter(Texture::MipMapNone);
    proxyTex->SetMagnificationFilter(Texture::MinMagNearest);
    proxyTex->SetMinificationFilter(Texture::MinMagNearest);
    proxyTex->SetWrapModeU(Texture::ClampToEdge);
    proxyTex->SetWrapModeV(Texture::ClampToEdge);

    ...

}
```

The result is that the content of the off-screen render target is displayed as texture in the display render target. See the screenshot below:



See also:

Multiple Render Targets for how to configure an offscreen render target in SceneComposer.

Example: External Image Source

External Image Source

This section shall be understood more as a proposal on how you might implement your own external image source than as tutorial. Reason is that various OpenGL ES extensions exist, a concrete tutorial therefore would be too extension specific.

Here a proposal is presented on how a class could be realized, that encapsulates a texture handle.

> Candera::ProxyTextureImage can be used with textures of type GL_TEXTURE_2D or GL_TEXTURE_CUBE_MAP. For other types, e.g. 3D textures, activation and other steps have to be done using own implementations along with node listeners.

Following class is a proposal on how an external image source could look like. Have a look at the comments to figure out the details.

```
class ExternalImageSource : public ImageSource3D       //First step is to derive from ImageSourc
{

public:
    ExternalImageSource();
    virtual ~ExternalImageSource();

    //Uploads the external image source to VRAM, ensure that the desired OpenGL ES Context is ac
    //Implementation has to be extension specific.
    bool Upload();

    //Unloads the external image source from VRAM, ensure that the desired OpenGL ES Context is
    //Implementation has to be extension specific.
    bool Unload();

    /*
 Updates the external image source in VRAM, ensure that the desired OpenGL ES Context is activat
 Adapt function to your needs.
 Has to be called whenever the image changes.
 Implementation has to be extension specific.
     */
    bool Update();

    //Implement ImageSource3D interface:
    virtual Handle GetVideoMemoryHandle() const;
    virtual bool IsMipMappingEnabled() const { return m_isMipMappingEnabled; }
    virtual Int GetHeight() const { return m_height; }
    virtual Int GetWidth() const { return m_width; }
    virtual void Sync() {}
    virtual void WaitSync() {}

    /*
 Getters and setters as well as members to interact with the external image source, such as a vi
 player, have to be added to this header.
     */

#ifndef GL_VIV_direct_texture
    void SetBitmap(const Bitmap::SharedPointer& bitmap) { m_bitmap = bitmap; }
    const Bitmap::SharedPointer& GetBitmap() const { return m_bitmap; }
#endif

private:

    bool m_isMipMappingEnabled;
    Int m_width;
    Int m_height;
    Handle m_videoMemoryHandle[CANDERA_MAX_CONTEXT_COUNT];

#ifdef GL_VIV_direct_texture
    GLvoid* m_texels;       //Direct pointer to buffer in VRAM!
#else
    Bitmap::SharedPointer m_bitmap;
```

```
#endif


    /*
 Getters and setters as well as members to interact with the external image source, such as a vi
 player, have to be added to this header.
     */

};
```

```
Handle ExternalImageSource::GetVideoMemoryHandle() const
{
    Handle handle = m_videoMemoryHandle[ContextResourcePool::GetActive().GetIndex()];
    return handle;
}
```

Finally, set the external image source as image source of your proxy texture. Upload should be called manually before or after uploading other nodes, but at least before your render loop starts.

Update should be called any time the external image source changes (e.g. a new frame is grabbed from a video). When calling Update, ensure that it is called before rendering the next camera frame.

```
    //An instance of the external image source (or pointer to) is needed.
    ExternalImageSource m_externalImage;
```

```
void ExternalImageSourceApp::InitScene()
{
    m_scene = Scene::Create();
    m_scene->SetName("Scene");

    m_light = Light::Create();
    m_light->SetType(Light::Directional);
    m_light->SetAmbient(Color(0.3f, 0.3f, 0.3f));
    m_light->SetDiffuse(Color(1.0f, 1.0f, 1.0f));
    m_light->SetSpecular(Color(1.0f,1.0f,1.0f));
    m_light->SetDirection(Vector3(0.0f, -5.0f, 0.0f));
    m_light->SetName("Light");
    m_light->SetRenderingEnabled(true);
    m_light->SetSpotAngle(30.0f);
    m_light->SetSpotExponent(30.0f);
    m_light->SetAttenuationEnabled(true);
    m_light->SetAttenuation(1.0f, 0.0f, 0.0f);
    m_scene->AddChild(m_light);


    //Create External image source, for demonstration with a bitmap.
    m_assetFactory.CreateBitmap(BitmapData_color_map.name, m_colorMapBmp);
    m_externalImage.SetBitmap(m_colorMapBmp);
```

```
    //Upload external image. Ensure context is active!
    m_gdu->ToRenderTarget3D()->Activate();
    m_externalImage.Upload();

    //Now create ProxyTextureImage.
    m_externalTextureImage = ProxyTextureImage::Create(&m_externalImage);

    //Create texture with proxy image.
    SharedPointer<Texture> texture = Texture::Create();
    texture->SetTextureImage(m_externalTextureImage);

    //Finally create node (billboard) to display proxy image.
    m_billboard = Billboard::Create(1.0f, 1.0f);
    m_billboard->SetAppearance(Appearance::Create());
    m_billboard->GetAppearance()->SetShader(m_refTransRefTexShader);
    m_billboard->GetAppearance()->SetMaterial(Material::Create());
    m_billboard->GetAppearance()->GetMaterial()->SetDiffuse(Color(1.0f, 1.0f, 1.0f, 1.0f));
    m_billboard->GetAppearance()->SetRenderMode(RenderMode::Create());
    m_billboard->GetAppearance()->SetShaderParamSetter(GenericShaderParamSetter::Create());
    m_billboard->GetAppearance()->SetTexture(texture,0);
    m_billboard->SetPosition(0.0f, 0.0f, -2.5f);
    m_billboard->SetRotation(0.0f, 0.0f, 0.0f);
    m_billboard->SetScale(1.5f, 1.5f, 1.5f);

    m_scene->AddChild(m_billboard);

}
```

```
{
    //You need to update your external image source before rendering.Again make sure that contex
    m_gdu->ToRenderTarget3D()->Activate();
    m_externalImage.Update();
}
```

```
void ExternalImageSourceApp::OnRender()
{
    m_gdu->ToRenderTarget3D()->Activate();
    Renderer::RenderAllCameras();
    m_gdu->ToRenderTarget3D()->SwapBuffers();
}
```

Multisample Anti-Aliasing Offscreen Render Targets

Description

This section describes how to set the multisample anti-aliasing (MSAA) property for offscreen render targets.

Anti-Aliasing and MSAA

Anti-Aliasing

In computer graphics, anti-aliasing is a technique used for improving image quality by smoothing sharp edges cause by aliasing. This usually occurs when rendering high-resolution signals at a lower resolution.

Multisample Anti-Aliasing

Multisample anti-aliasing is a method used for full-screen anti-aliasing by multisampling each pixel at multiple pixel locations. The fragment shader is run only once per pixel no matter the numbers of samples covered by the rendered polygon. Once the color buffer's samples are calculated, these colors are averaged per pixel to produce the final color. Depth and stencil values also make use of the multisample points which means that their buffer size will increase by the amount of samples per pixel.

The higher the number of additional samples per pixel will result in an increase in the amount of anti-aliasing which will generate even smoother edges. The application will noticeably suffer in terms of performance the more samples are used.

There are four modes for MSAA: 2x, 4x, 8x, and 16x, but out of these, 4x MSAA offers a good balance between performance and quality.

> MSAA for offscreen render targets is only available on devices that support OpenGL ES 3.0 or higher. Even more, the number of supported samples differs from one device to another, some may not support more than 4x MSAA, which is the minimum defined by the OpenGL ES 3.0 standard. Using sample values higher than supported might lead to undefined behaviours or errors. It is recommended to check the device documentation for the number of supported samples.

MSAA Offscreen Render Targets

Setting MsaaSamples Property

Enabling of multisample anti-aliasing with a given number of samples is achieved by setting the MsaaSamples property of a framebuffer object to a value larger than 1 (default value).

The MsaaSamples property values is changed either using Candera::GlFrameBufferProperties::SetMsaaSamples method or by meta information.

The following snippets show how to set MsaaSamples using the methods described above:

Using Candera::DevicePackageInterface

- create an offscreen render target using Candera::DevicePackageInterface

```
    m_offscreenRenderTarget = Base::CreateGraphicDeviceUnit(DevicePackageDescriptor::OffscreenTa
    if (m_offscreenRenderTarget == 0) {
        return false;
    }
```

- access render target properties (including MsaaSamples) via MetaInfo

```
    const MetaInfo::GraphicDeviceUnitMetaInfo *meta = DevicePackageDescriptor::GetMetaInformatic
    meta->LookupItem("Width") && meta->LookupItem("Width")->Set(m_offscreenRenderTarget, m_rtPro
    meta->LookupItem("Height") && meta->LookupItem("Height")->Set(m_offscreenRenderTarget, m_rtP
    meta->LookupItem("ColorFormat") && meta->LookupItem("ColorFormat")->Set(m_offscreenRenderTar
    meta->LookupItem("DepthFormat") && meta->LookupItem("DepthFormat")->Set(m_offscreenRenderTar
    meta->LookupItem("MsaaSamples") && meta->LookupItem("MsaaSamples")->Set(m_offscreenRenderTar

    if(m_offscreenRenderTarget->Upload() != true){
        FEATSTD_LOG_ERROR("Offscreen render target upload failed\n");
        return false;
    }
```

Using Candera::iMX6FrameBufferObject

- declare an iMX6 framebuffer render target

```
            iMX6FrameBufferObject m_fbo;
            iMX6FrameBufferObject* m_frameBufferRenderTarget;
```

- access render target properties using their corresponding setter methods

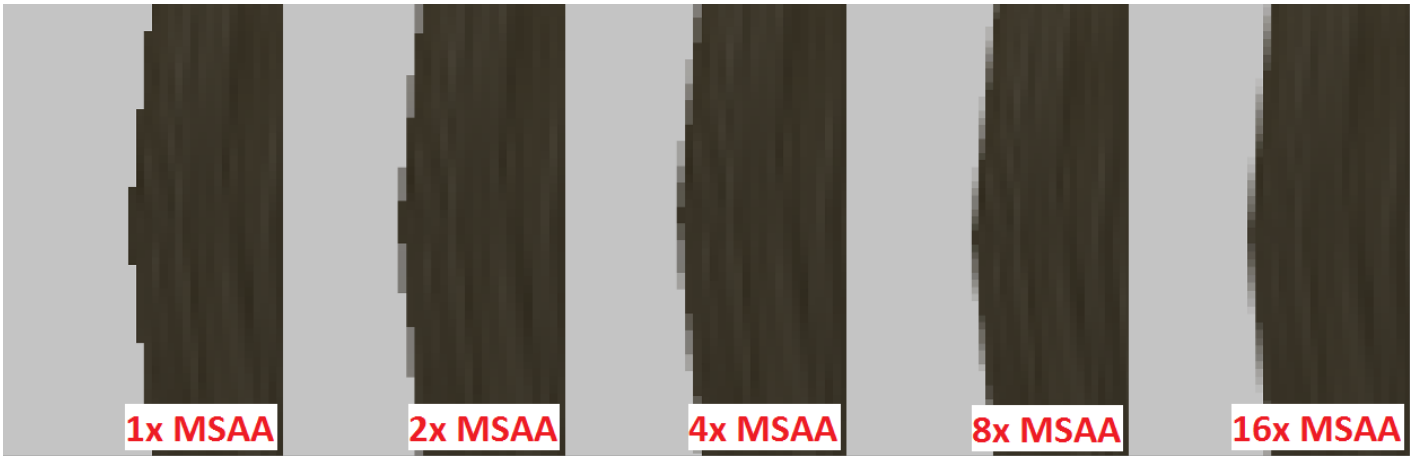
```
    m_frameBufferRenderTarget = &m_fbo;
    m_frameBufferRenderTarget->GetProperties().SetHeight(c_fbHeight);
    m_frameBufferRenderTarget->GetProperties().SetWidth(c_fbWidth);
    m_frameBufferRenderTarget->GetProperties().SetMsaaSamples(c_msaaSamples);
    m_frameBufferRenderTarget->Upload();
```

Framebuffer Blit

These multisample renderbuffers cannot be directly bound to textures. A solution is provided by OpenGL ES 3.0 by using the framebuffer blit function, such that these renderbuffers are resolved to single-sample textures with the use of an intermediate framebuffer. The blit function is responsible to transfer a region defined by 4 screen-space coordinates of the read framebuffer to another region in the draw framebuffer, turning the multisample buffer into a 2D texture that could be further used for post-processing in the fragment shader.

Example: MSAA result for different sample values

The results of multisample anti-aliasing for 1x - default, 2x, 4x, 8x and 16x sample rates can be seen in the image below.



External Texture Image

How to use EGLKHRExternalTexture

To use an DMA buffer as an external texture create an array to hold necessary information. This varies with platforms and types of buffers. This example shows using DMS buffers on RCarH3.

```
//Activate appropriate RenderTarget first.

EGLint defaultAttribs[13];
defaultAttribs[0] = EGL_DMA_BUF_PLANE0_FD_EXT;
defaultAttribs[1] = <file descriptor of the buffer>;  //file descriptor
defaultAttribs[2] = EGL_DMA_BUF_PLANE0_OFFSET_EXT;
defaultAttribs[3] = 0; // depends on platform and requirements
defaultAttribs[4] = EGL_DMA_BUF_PLANE0_PITCH_EXT;
defaultAttribs[5] = <pitch of the buffer>;
defaultAttribs[6] = EGL_LINUX_DRM_FOURCC_EXT;
defaultAttribs[7] = DRM_FORMAT_ARGB8888;  // depends on platform and requirements
defaultAttribs[8] = EGL_WIDTH;
defaultAttribs[9] = 255; // width of the image. Also set this to the EglKhrExternalTextureImage
defaultAttribs[10] = EGL_HEIGHT;
defaultAttribs[11] = 456; // height of the image. Also set this to the EglKhrExternalTextureImag
defaultAttribs[12] = EGL_NONE; //Terminator
Candera::EglKhrExternalTextureImage::SharedPointer ptr =
Candera::EglKhrExternalTextureImage::Create();
ptr->SetRole(<Candera::EglKhrExternalTextureImage::Consumer| Candera::EglKhrExternalTextureImage
ptr->SetAttributeList(defaultAttribs);  // will be deep copied.
ptr->SetExternalBufferType(EGL_LINUX_DMA_BUF_EXT); // buffer type
ptr->SetExternalBuffer(static_cast<EGLClientBuffer>(0)); // actual buffer
ptr->SetWidth(255);
ptr->SetHeight(456);
ptr->Upload();
```