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.
}
```

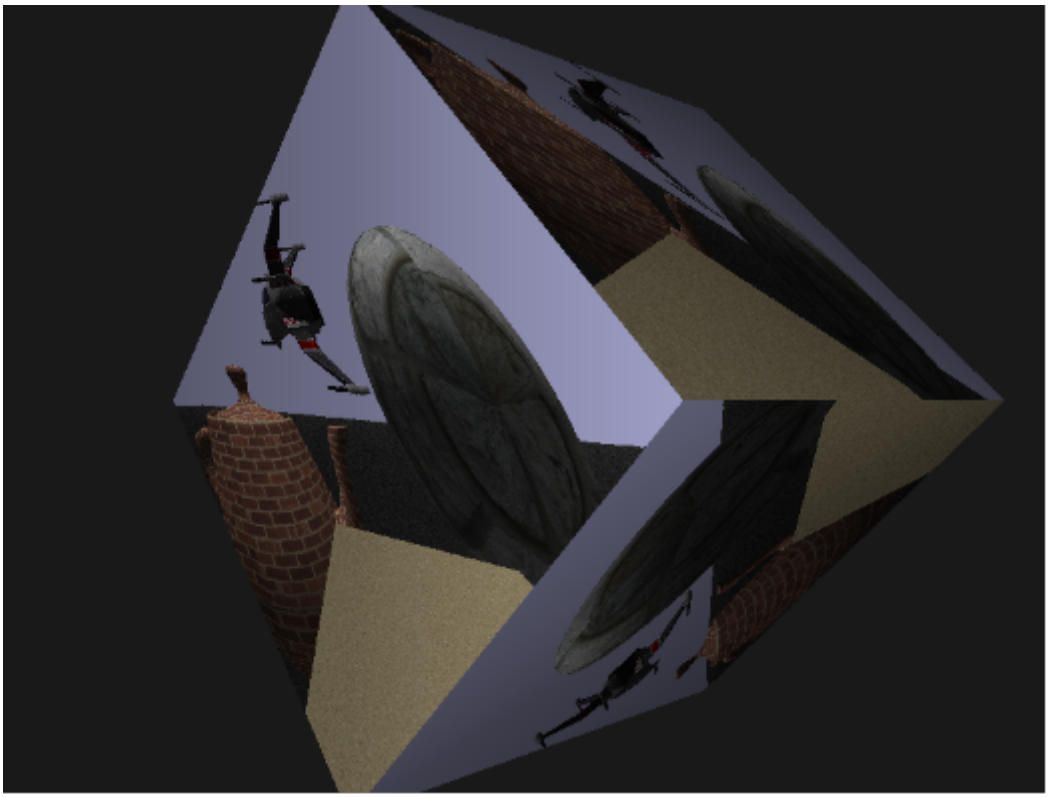
Finally, set the off-screen render target as image source of the proxy texture.

```
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();
}
```

Revision #7
Created 2023-02-22 08:35:58 UTC by Tsuyoshi.Kato
Updated 2025-01-17 08:52:51 UTC by Elisabeth Zauner