Tutorial for Shader Usage

- Programmable Graphics Pipeline
- The OpenGL ES Shading Language
- Shader Usage in Candera
- Candera Reference Shaders

Programmable Graphics Pipeline

Description

When working with shaders, a fundamental knowledge of OpenGL ES 2.0 / 3.0 and the corresponding GLSL ES 1.0 / 3.0 shader language is required, to unleash the freedom and power programmable graphic pipelines provide, compared to the former fixed-function graphics pipeline. The focus of this chapter is to give the reader a basic overview of the OpenGL ES 2.0 / 3.0 rendering pipeline, in order to understand how OpenGL ES 2.0 / 3.0 works, and how the advantages that come with shader programming can be leveraged.

OpenGL ES 2.0 / 3.0 Graphics Pipeline

The OpenGL ES 2.0 / 3.0 Graphics Pipeline determines how data stored in Vertex Buffers and Textures get processed, in order to produce the final image in the framebuffer (which might be displayed on screen or an off-screen render buffer, see Tutorial 7).

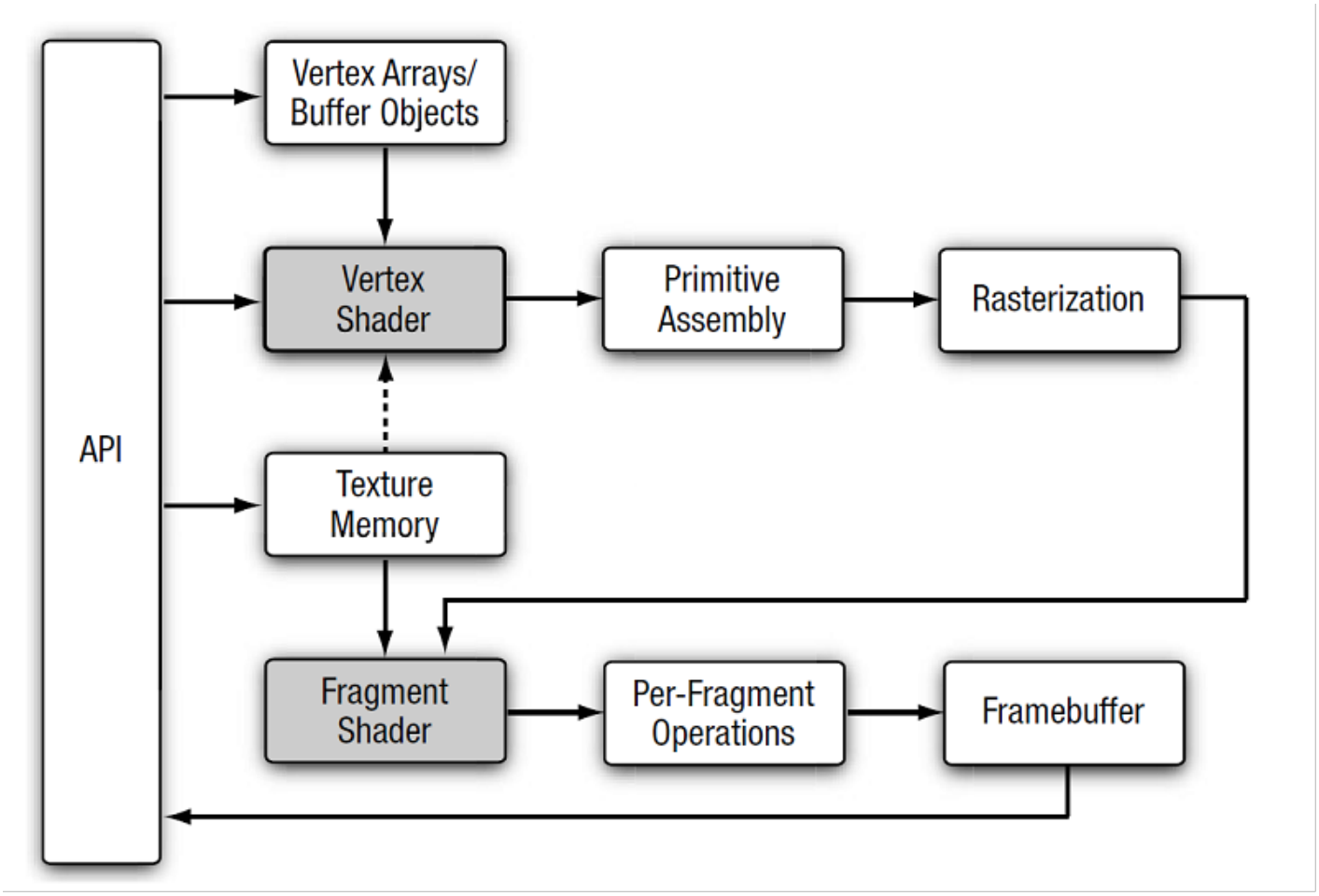
> Only a simplified view is described here, in order to enable the reader to gain a basic knowledge and comprehension of shader programs, their purpose and principles of operation.

Basically the data passed to the OpenGL ES 2.0 API (encapsulated by Candera::Engine3D::Core Classes, such as Candera::VertexBuffer, Candera::Texture) is stored in **Vertex Buffers** and **Texture Memory**.

Vertex Buffers get processed by the **Vertex Shader unit**, and are transformed and projected from model space into world space, and finally screen space. This data is then assembled into primitives (triangles, lines, points) which subsequently get rasterized to determine which pixels are covered by the primitive and are subject to be rendered.

> Pixels at this stage of the pipeline, are known as fragments, as they might be discarded by the pipeline in subsequent tests and thus never appear as a pixel in color buffer.

The **Fragment Shader** then takes these fragments, interpolates the attributes of each vertex inside a primitive and combines these with the colors stored in **Texture Memory**. After applying some tests, blending and dithering, the final pixel gets written to the **Framebuffer**. The following image describes this workflow graphically.[2]



The following table shortly describes the single stages in the image above.

Stage	Description
API	OpenGL ES application programming interface, this is encapsulated by Candera.

Vertex Arrays / Buffer Objects	Buffer in VRAM that stores vertices along with their vertex attributes. Vertex attributes are typically positions, normals, texture coordinates, colors etc. which are stored for every vertex of every primitive. This array is filled and managed by Candera::VertexBuffer.
Vertex Shader	Processes the data stored in vertex buffer, and transforms them to screen coordinates with depth.
Primitive Assembly	Assembles the data processed shaded vertices into individual geometric primitives that can be drawn. Such primitives can be triangles, lines or points. The necessary settings here are also set in Candera::VertexBuffer.
Rasterization	Single primitives are converted to a set of two-dimensional fragments (describing the normalized position in the framebuffer). This is the input data to the fragment shader.
Texture Memory	Texture memory is the location in VRAM where textures (2D or CubeMaps. In OpenGL ES 2.0, 1D and 3D textures are only supported through extensions) are stored. This memory is filled and managed by Candera::Texture.
Fragment Shader	Fragment shader takes the processed fragments as input, and combines them with the data stored in Texture memory. This stage results in assigning the fragment a color.
Per-Fragment Operations	Each fragment is tested, to determine whether it is to be drawn. Blending and dithering operations conclude the pipeline. Various configurations are set, using Candera::RenderMode. Here Depth-, Stencil-, Scissoring Tests, Blending and Dithering apply.
Framebuffer	Includes color, depth-, and stencil buffer to store final render output.

This chapter described the stages **Vertex Shader** and **Fragment Shader**. The next chapter explains how to interact and control programmable shaders.

Programmable Shaders

Description

This chapter is about user-programmable Vertex and Fragment Shader units.

Chapters

- Shader Programs Introduction
- Vertex Shaders

- Fragment Shaders

Shader Programs Introduction

Programmable Shaders

Before OpenGL 2.0 and OpenGL ES 2.0, Vertex Shader and Fragment Shader units had fixed functionality. The common wording for this is "fixed function pipeline". Since OpenGL 2.0 and OpenGL ES 2.0 the behavior of these shaders became user-programmable. The wording is "programmable pipeline".

Being able to program these stages of the pipe offers a whole new set of opportunities. The shaders still have the same inputs and outputs, but the way in which these outputs are generated can now be defined by the user. Also, using user-defined variables may now be passed to the shader.

Programmable **Vertex Shader** units offer the possibility of manipulating every single vertex of geometry in a user-defined way, using different input attributes, user-defined uniform variables or even textures to compute the final outcome.

Fragment operations allows the final color to be computed from different sources, as the user wishes. This gives the user the powerful ability to produce almost every desired effect. Typical examples range from per pixel-lighting, normal mapping, multi-texturing, environment mapping (using cubemaps) and many more.

However, there is one small disadvantage. The "fixed-function" pipeline has been replaced by the programmable one, which forces the user to re-implement any behaviour of the "fixed function pipeline", which may be required.

Shader Programs

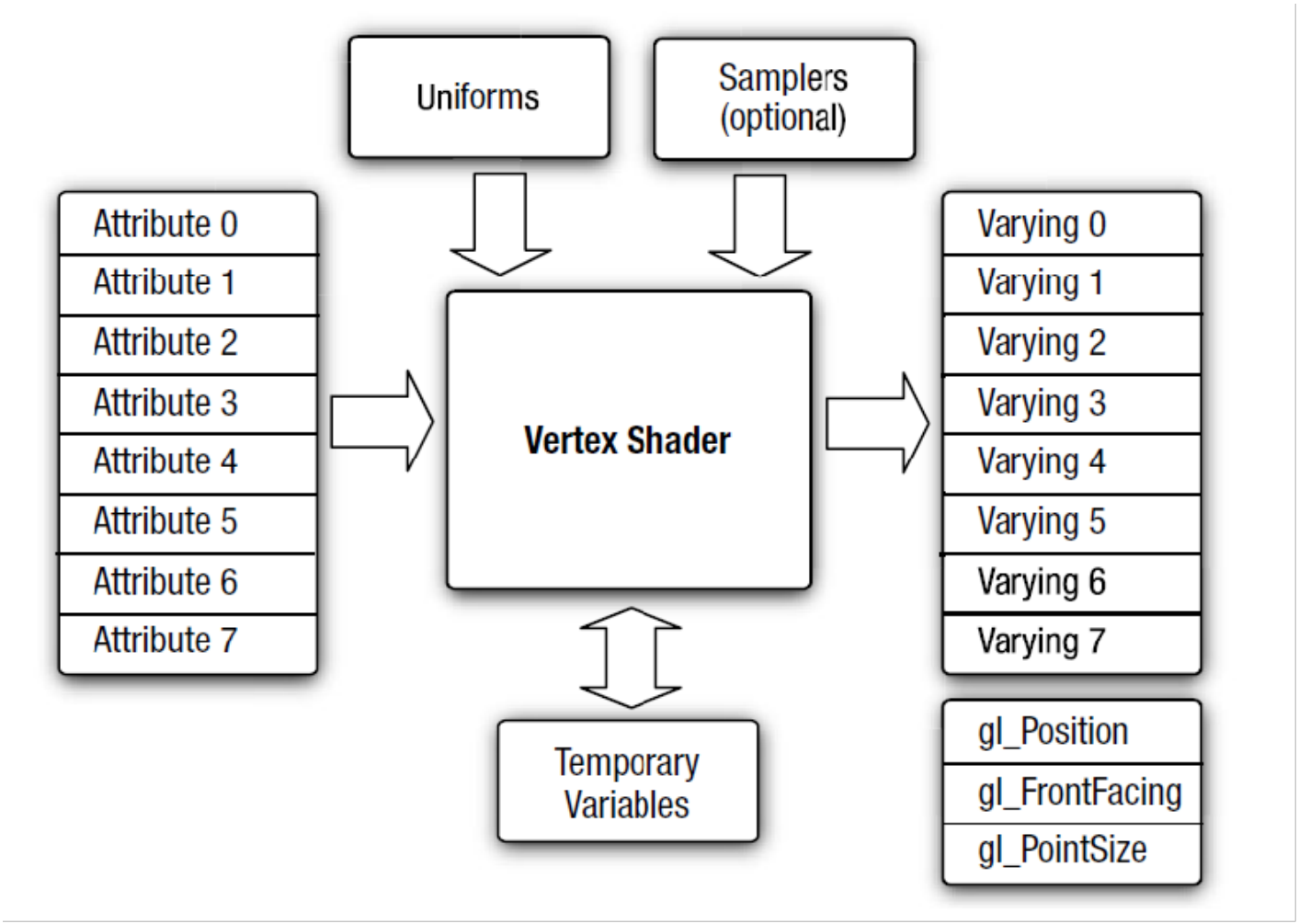
A shader program is the user-generated code that gives the pipeline its user-defined behavior. A user of OpenGL ES 2.0 / 3.0 has to provide this before rendering any object. In Candera this is encapsulated in Candera::Shader as described in Tutorial 3. A shader program consists of two parts, a vertex shader and a fragment shader, which must be linked together to form the shader program.

These two shader objects are generated from a special programming language known as the OpenGL Shading Language (ES) or GLSL (ES), which uses a C-like syntax. The method in which the shader is passed, is platform dependent. On Desktop GPU's the shader compiler is normally integrated into the graphics driver, and therefore the GLSL source is directly passed to Candera::Shader::SetVertexShader and Candera::Shader::SetFragmentShader, while on some target devices, pre-compiled binaries are expected.

For a detailed step-by-step tutorial see the corresponding section. What follows is an overview on how the different types of shader objects work and which data they process.

Vertex Shaders

Vertex shader objects are the program objects that give Vertex Shaders user-defined behavior. The following image illustrates the input and output data processed by this kind of shader object..[3]



Attributes are the data components of a vertex, describing values such as position, normal, texture coordinates, color, tangent or binormal. This data is stored inside the vertex buffer and is the input that comes over the OpenGL ES 2.0 / 3.0 pipeline. The vertex shader is executed once for each incoming vertex.

Uniforms and **Samplers** (actually also passed as uniform) are user-defined input variables that are passed to every instance of a shader. A user defines it once for a shader object, but all shaders get this value. This input can consist of 4x4-, 3x3-, 2x2- matrices, 2-4 component vectors, single values and arrays, all of type Float, Int or Bool. Special sampler uniforms are handles to uploaded 2D- or CubeMap Textures. Textures can also be used in vertex shaders, a good example for this is Displacement Mapping, where the values stored in the textures colors are interpreted as vectors to offset the vertices. This can be used e.g. to transform a plane into a virtual terrain.

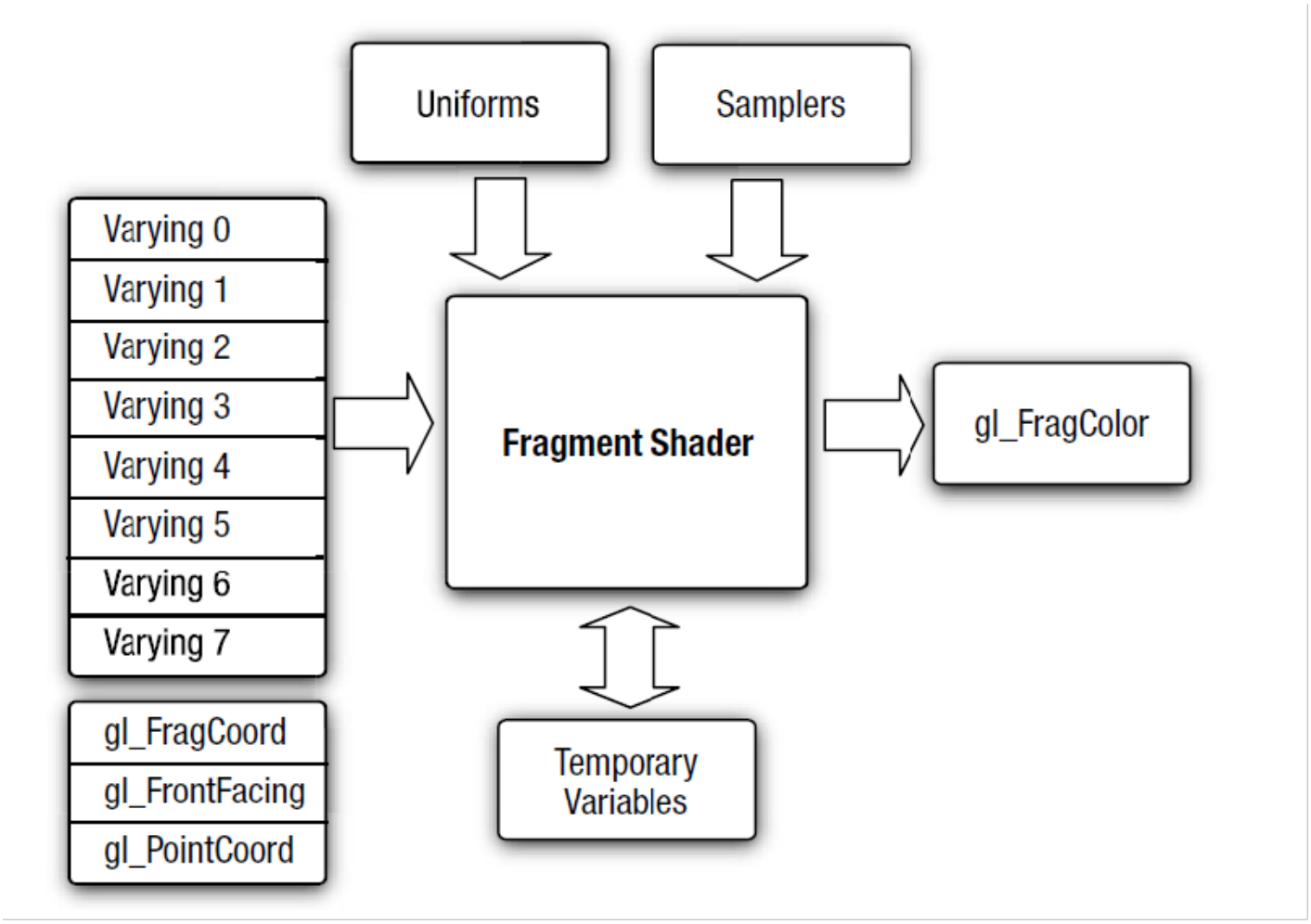
Varyings are user-defined output variables of a vertex shader, which are finally interpolated between all members of a primitive after primitive assembly. These can have the same data types as the incoming uniforms, except for sampler handles. For each fragment, the interpolated data then gets passed into the fragment shader. Typically texture coordinates are forwarded from the vertex buffer to the fragment shader in this way, but you can pass any kind of data where it makes sense to interpolate between the vertices of a primitive.

Temporary Variables are variables which live inside the scope of one shader object. They can have the same data types as uniforms or varyings.

Vertex Shader Special Output Variables *gl_Position* and *gl_PointSize* are only available in vertex shaders. *gl_FrontFacing* is not available in OpenGL ES, but only in full OpenGL. *gl_Position* is reserved for writing the vertex position in homogeneous coordinates (after transforming into world space, camera space and projected to screen space). This value has to be written by every vertex shader, in order to enable primitive assembly and rasterization. Not setting *gl_Position* results in undefined behavior. *gl_PointSize* can be used optionally to set the point size of vertices, if the primitive type points is used.

Fragment Shaders

Fragment shader objects are the program objects that give Fragment Shaders user-defined behavior. The following image illustrates the input and output data processed by this kind of shader object.[4]



Varyings are user-defined input variables for a fragment shader. These values have to match the varyings in the vertex shader (data type and name). A fragment shader is called once for each rendered pixel, which implies that the incoming varyings hold the interpolated value of all varyings passed per primitive from the vertex shader.

Uniforms and **Samplers** are user-defined input variables that are passed to every instance of a shader. They have the same meaning as in the vertex shader. Additionally uniforms in vertex and fragment shaders can have the same data type and name. In this case, when passing the variables value, it'll be passed to both the vertex and the fragment shader.

Temporary Variables are variables which live inside the scope of one shader object. They can have the same data types as uniforms or varyings.

Fragment Shader Special Input Variables *gl_FragCoord*, *gl_FrontFacing* and *gl_PointCoord* are read-only variables only available in fragment shaders. *gl_FragCoord* holds the window relative coordinates x,y,z and 1/w values for the fragment. Z-value is the value that's used for depth testing after passing the Fragment Shader stage. *gl_FrontFacing* holds a boolean value which determines whether the fragment belongs to a front facing primitive (true) or not (false). *gl_PointCoord* specifies where the fragment is located, within a point primitive. If the primitive is of type line or triangle, this value is undefined.

Fragment Shader Special Output Variable *gl_FragColor* is used to write the output of the fragment shader into the current fragment. *gl_FragColor* holds a 4-component vector of floats ranging from 0.0 to 1.0 for each component. This describes the RGBA color of the current fragment, used by the rest of the OpenGL ES 2.0 / 3.0 rendering pipeline. This will be the color you actually see on the screen or will be blended against another color, if the fragment passes all succeeding tests. Not setting this value will result in undefined colors on the screen.

References

The knowledge presented in this tutorial is a summary of:

[1] Munshi, Aaftab; Ginsburg, Dan; Shreiner, Dave: OpenGL ES 2.0 Programming Guide. Addison Wesley, 2008. ISBN 978-0-321-50279-7

The images were taken from:

[2] Munshi, Aaftab; Ginsburg, Dan; Shreiner, Dave: OpenGL ES 2.0 Programming Guide. p.4. Addison Wesley, 2008. ISBN 978-0-321-50279-7

[3] Munshi, Aaftab; Ginsburg, Dan; Shreiner, Dave: OpenGL ES 2.0 Programming Guide. p.5. Addison Wesley, 2008. ISBN 978-0-321-50279-7

[4] Munshi, Aaftab; Ginsburg, Dan; Shreiner, Dave: OpenGL ES 2.0 Programming Guide. p.8. Addison Wesley, 2008. ISBN 978-0-321-50279-7

The OpenGL ES Shading Language

Description

The focus of this chapter is to understand how to write shader programs in conjunction with Candera. In OpenGL ES 2.0 / 3.0, the OpenGL ES Shader Language (GLSL ES) is used as programming language for creating various effects, from simple texturing to stunning effects, like bump mapping, anisotropic lighting, and so forth. The programmable pipeline of ES 2.0 is also valid for ES 3.0, but the shading language has been extended somewhat. OpenGL ES 2.0 uses the GLSL ES 1.0 shading language. OpenGL ES 3.0 uses the GLSL ES 3.0 shading language.

For more detailed information you may refer to the GLSL ES specification or to the OpenGL ES 2.0 Programming Guide. Additionally, vast resources of GLSL as well as GLSL ES shader tutorials can be found on the internet. GLSL is to a large extent compatible to the GLSL ES shader language.

See also:

For hints on how to optimize shaders refer to

- Optimizing Shader Programs in **Application Development Best Practices**

A Minimal Shader Program

Here is a small example to demonstrate the syntax of GLSL ES. The following example simply transforms the vertices into clip coordinate space, forwards texture coordinates, and applies a texture to the rasterized image.

Vertex Shader

This is the vertex shader code of our example:

```
#ifndef GL_ES

// An OpenGL environment might not support precision qualifiers. Thus, define them to void.
```

```
#define lowp
#define mediump
#define highp
#define precision

#endif

/*
 Uniforms
 */
uniform mat4 u_MVPMatrix;       // Model-View-Projection Matrix

/*
 Attributes
 */
attribute vec4 a_Position;
attribute vec2 a_TextureCoordinate;

/*
 Varyings
 */
varying mediump vec2 v_TexCoord;

/*---------------------------------- MAIN ------------------------------------*/
void main(void)
{
    /* Transform vertex into world space */
    gl_Position = u_MVPMatrix * a_Position;

    v_TexCoord = a_TextureCoordinate;
}
```

Purpose of this vertex shader is simply to transform the incoming *a_Position* vertex attribute into clip coordinate space, using the model-view-transformation matrix provided as uniform *u_MVPMatrix*. Additionally, the incoming attribute *a_TextureCoordinate* is forwarded to the next stage as varying *v_TexCoord* and will be available in the fragment shader, if a varying with same name exists. In the fragment shader the incoming value will be interpolated between the vertices of a primitive.

The preprocessor directives (#ifndef #define ... #endif) simply check if the shader is running inside a OpenGL ES or OpenGL rendering pipeline and defines the precision qualifiers to nothing if not running in an ES context. This simply transforms the OpenGL ES shader into a compatible OpenGL shader if necessary, as the only differences are the precision qualifiers.

Fragment Shader

This is the fragment shader code of our example:

```
#ifndef GL_ES

// An OpenGL environment might not support precision qualifiers. Thus, define them to void.
```

```
#define lowp
#define mediump
#define highp
#define precision

#endif

/*
 Uniforms
 */
uniform sampler2D u_Texture;

/*
 Varyings
 */
varying mediump vec2 v_TexCoord;

/*---------------------------------- MAIN ------------------------------------*/
void main(void)
{
    gl_FragColor = texture2D(u_Texture, v_TexCoord);
}
```

Purpose of this fragment shader is to query the color from the passed sampler *u_Texture* (handle to a texture) at the address of the incoming varying *v_TexCoord* and to apply this color to the fragment. This will be the color that is finally displayed, if all succeeding tests have passed. Nothing more, nothing less. It can be observed that the varying variable *v_TexCoord* has the same name than in the vertex shader.

The preprocessor fulfills the same purpose as in the vertex shader.

Output

The following image shows the result of rendering a vertex buffer (containing vertices of a cube) using the vertex and fragment shader above. The model-view-projection matrix was set to rotate the cube, a 2D texture was applied to it as sampler2D u_Texture.



Basic OpenGL ES 2.0 Shading Language elements

Basic OpenGL ES 2.0 / 3.0 Shading Language elements

The subject of this section is to comprehend the syntax and structure of the vertex and fragment shaders, presented in the lean shader example, see the previous A Minimal Shader Program. Knowledge of C like languages is assumed to understand GLSL syntax.

Declaring Uniform variables

The qualifier *uniform* is used to declare immutable variables in vertex or fragment shaders, which are set from client code.

> Uniforms are global variables, and therefore have to be declared outside of function scope.

In our example the following uniforms are used:

```
uniform mat4 u_MVPMatrix;
```

u_MVPMatrix is used to pass the model-view-projection matrix to the vertex shader object, which is used to transform all vertices.

```
uniform sampler2D u_Texture;
```

u_Texture is the handle to the texture, which is addressed using the texture2D function.

Declaring Varying variables

The qualifier *varying* is used in two different ways.

- In the vertex shader it is used to declare variables which are written by the shader, and are further passed over primitive assembly and rasterization. Therefore they get interpolated inside the fragments, which are finally input of the fragment shader.
- In fragment shader the same keyword is used to declare incoming variables. Like uniforms these varying variables are global, and therefore have to be defined outside function scope.

> Varying variables have to have exactly the same name and type in vertex and fragment shader, otherwise linking of them together will fail.

In our example only one varying is used:

```
varying mediump vec2 v_TexCoord;
```

v_TexCoord is used to pass the texture coordinates stored inside the vertices attributes to the fragment shader. While passing the OpenGL ES 2.0 pipe they get interpolated between the vertices of a primitive. Inside the fragment shader the interpolated coordinates are finally used to address the texture sampler, in order to determine the final color of a fragment.

Precision Qualifiers

Compared to "full" GLSL, GLSL ES has the specialty of precision qualifiers, which can be used to declare variables to be handled with different precision. Reason for this language feature was the focus on embedded hardware, which might provide faster medium precision floating point or integer units than the high precision ones. Precision of a float or uniform variable is defined using one of the qualifiers *lowp*, *mediump* or *highp*.

For further details on how these kinds of variables behave when they get assigned to each other, or when arithmetic is used with different precision, please refer to the GLSL ES standard.

In our example this was used for the varying variable in both shaders:

```
varying mediump vec2 v_TexCoord;
```

Function Declaration

Functions can be defined like in almost every other C-like programming language. Function definitions follow the syntax shown below:

```
returnType functionName (type0 arg0, type1 arg1, ... , typen argn)
{
    //Method Body
    return returnValue;
}
```

Like in C they can be prototyped with leaving the method body and closing the declaration using a semicolon. For calling the function of course it has to be defined.

A function then is called using its name followed by a list of arguments in parentheses (or empty ones if no arguments apply).

Function names can further be overloaded as long as the argument types are different.

Return and argument types can be any type allowed in GLSL (such as int, float, double, vec4, mat4, sampler2D, ...). Additionally returnType can be void, which indicates that the function doesn't return anything. Arrays can be passed as argument, but cannot be returned.

In our examples above only one function is defined for the vertex and the fragment shader:

```
void main(void)
{
    //Main Body
}
```

The main function is mandatory for both vertex and fragment shader, this is the entry point to the shader. It has to take no argument and has to return no value (void). From here further functions can be called.

Writing Output Data

There are two possibilities to output a **vertex shaders** result.

- The first and mandatory way is to write the transformed vertex coordinates in clip coordinate space to *gl_Position*. This one is required to be set by any vertex shader. The behavior is undefined, if *gl_Position* is not set.
- The second, optional way is to use variables declared as varying to output any desired data, as described in the varying section. The content of varyings is passed as input to the

fragment shader, the fragment shader will receive the interpolated values per transformed primitive.

Results of **fragment shaders** are written to *gl_FragColor*, which is a 4-component vector (vec4) representing the final color that will be displayed if all further tests pass. It is possible to use the *discard* keyword to abort a fragment shader based on conditions, which might be useful if some fragments shall be completely skipped, in this case also no depth or stencil value will be written. This trick can be applied for e.g. creating an interlace mask for interlaced stereoscopic 3D.

Accessing textures

One very important function of fragment shaders is to fetch colors from textures and use it for composing the final *gl_FragColor*. This task is done using the GLSL ES built-in function *texture2D*:

```
vec4 texture2D (sampler2D sampler, vec2 coord );
```

where *sampler* is the handle to the texture to query the color from, *coord* is the texture coordinate pair to address a certain "texel" (point of a texture).

The *sampler2D* variable has to be passed as uniform to the shader, this is the OpenGL handle to the texture in VRAM. In our simple fragment shader example the color read from the texture is passed directly as *gl_FragColor*, therefore the resulting cube has the wall image directly applied.

The following snippet shows how the uniform containing the texture handle is declared.

```
uniform sampler2D u_Texture;
```

The following incoming varying vec2 is used to address a texel.

```
varying mediump vec2 v_TexCoord;
```

The final color is queried from the texture using the *texture2D* function.

```
gl_FragColor = texture2D(u_Texture, v_TexCoord);
```

> Texturing can also be used in vertex shaders. This can be used to e.g. realize a technique called *Displacement Mapping* where the vectors stored inside a texel are not interpreted as color but as offset being applied to the current vertex position. Consequently, a displacement map can be used e.g. to transform a flat plane to a terrain.

Also other types of samplers (e.g. samplerCube) and corresponding texture addressing functions (e.g. textureCube) exist, again for more details please have a look at the GLSL ES Standard.

Commenting shader programs

Comments in GLSL are delimited, as in other C-like languages, by

```
/* and */
```

or

```
 // and EOL.
```

Everything in between these delimiters will be ignored by the compiler.

Other language features, data types and function library.

For a more detailed view on how to specify control structures, available data types, built-in functions, and other language features please refer to the GLSL ES Standard.

Shader Usage in Candera

Description

This chapter demonstrates how customized shader programs can be used with Candera. Subject of this chapter depicts how to use the Candera classes to set custom shaders for each node, and how to apply shader parameters. Further, it shows how to realize different classes of effects, namely single pass-, local multi pass- and global multi pass effects.

Shader Related Class Usage

This section is about on how the already introduced Candera::Appearance class and its members can be used to realize customized shader effects.

Loading shader programs

First step to realize shader programs is to actually write them. See the GLSL chapter for details on that. For a developer it is recommended to develop using SceneComposer, where the effects can be seen immediately in the Scene Editor window.

This tutorial further focuses on programming with Candera, but the concepts are supported as well in collaboration with SceneComposer. Beside of SceneComposer's shader editor, shader authoring tools like AMD's RenderMonkey can also be used. Benefits from using either SceneComposer or shader authoring tools are syntax highlighting, seeing compiler and linker errors and having a preview scene where the effect can be examined. One unique advantage of using SceneComposer is the support for device dependent shader compilers.

After having finished your shader program you need to load it in your application based on Candera . Use your favorite file I/O libraries among the available ones. It depends on the platform if the underlying OpenGL ES implementation needs the shader source or pre-compiled shader objects. On host you typically load the source of your shader program directly, it will be compiled and linked at runtime by the OpenGL driver invoked by Candera's Upload mechanisms. On targets you might need to provide pre-compiled shader objects, where pre-compiling is done using dedicated shader compile tools. Please refer to the device manufacturers manual.

Using Candera::Shader

Candera::Shader is the class that abstracts from OpenGL ES shader interfaces and manages the shaders handle after creation. For a small description see the tutorial on the Appearance class. See

below how to use the class.

First you need to create an instance of the Candera::Shader class using its Create() method.

```
static MemoryManagement::SharedPointer<Shader> Create();
```

Next step is to provide the loaded shader sources or objects to the class using the according setters:

```
bool SetVertexShader(const void* vertexShader, DisposerFn disposerFn);
bool SetFragmentShader(const void* fragmentShader, DisposerFn disposerFn);
```

> The shaders are passed as *void** to ensure that platform dependent representations of the shader program can be passed. The parameter DisposerFn hands over lifetime responsibility of shader passed: If it's set to 0 you'll have to take care of the provided *void** by yourself.

Finally, the shader needs to be associated to a node's appearance in order to leave the remaining steps (like Uploading and further Activating) to Candera. Upload takes care of compiling, linking, and retrieving the shaders handle. Vertex attribute binding is done in the appearances activation method, automatically.

The following code snippet demonstrates how a Billboard node with a custom shader can be created.

```
void Initialize() {
    Char* vertexShader = "";
    Char* fragmentShader = "";
    // Do File I/O here, filling vertexShader and fragmentShader with the read data.

    Scene* scene = Scene::Create();
    //Do camera setup here, add it to scene graph.

    SharedPointer<Shader> shader = Shader::Create();
    shader->SetVertexShader(vertexShader, 0); //vertexShader needs to be disposed manually.
    shader->SetFragmentShader(fragmentShader, 0); //fragmentShader needs to be disposed manually

    Billboard* billboard = Billboard::Create(2.0f, 2.0f);
    billboard->SetAppearance(Appearance::Create());
    billboard->GetAppearance()->SetShader(shader); //Here the previously instantiated and config
    //Set and configure remaining appearance members.

    scene->AddChild(billboard);
```

```
    scene->UploadAll(); //All nodes, all their Appearance objects and therefore all set shader c

}

void Render() {
    Renderer::RenderAllCameras(); //Here the scene graph gets traversed, for every node the shac
                                  //to the shaders attributes.
}
```

Auto Uniforms

Candera::ShaderParamSetter and derived classes are used to set uniform parameters of shaders, they abstract from OpenGL ES's glUniform_ interfaces. They are part of a node's appearance, how to use them is already described in the Appearance Tutorial.

As described the derived class Candera::GenericShaderParamSetter offers to automatically compute and set shader parameters from the configured scene graph. The following table shows which uniforms are reserved for this purpose, what Candera classes they affect and what their semantics are.

Note:
The values of reserved variables cannot be edited in SceneComposer, here the Candera::GenericShaderParamSetter has to be used.

The table is structured as follows:

- Column **Uniform Variable** is the name of the reserved variable.
- Column **Data Type** describes its data type.
- Column **GenericShaderParamSetter Flag** describes which properties have to be enabled in Candera::GenericShaderParamSetter in order to compute and set the variable.
- Column **Affected Classes** describes which classes are used to compute the parameters.
- The last parameter **Semantic** describes for what purpose this parameter is needed.

> If no instance of the affected classes exist, the corresponding shader parameters will not be set, even if they are enabled in Candera::GenericShaderParamSetter.

Uniform Variable	Data Type	GenericShaderParamSetter Flag	Affected Classes	Semantic

u_MMatrix3	Matrix3	ModelMatrix3Enabled	Candera::Node	Computes the ModelMatrix3 from the Node's transformation data, used to transform 3-component vectors from model to world space.
u_MVMatrix3	Matrix3	ModelViewMatrix3Enabled	Candera::Node, Candera::Camera	Computes the ModelViewMatrix3 from the Node's transformation data and the currently rendered Camera's ViewMatrix. Used to transform 3-component vectors from model to view space.
u_NormalMMatrix3	Matrix3	NormalModelMatrix3Enabled	Candera::Node	Computes the NormalModelMatrix3 from the Node's transformation data, used to transform 3-component directional vectors (especially normals) from model to world space.
u_NormalMMatrix4	Matrix4	NormalModelMatrix4Enabled	Candera::Node	Computes the NormalModelMatrix4 (4×4 normal-transform matrix) from the node’s transform and passes it to the shader.
u_NormalMVMatrix3	Matrix3	NormalModelViewMatrix3Enabled	Candera::Node, Candera::Camera	Computes the NormalModelViewMatrix3 from the Node's transformation data and the currently rendered Camera's ViewMatrix. Used to transform 3-component directional vectors (especially normals) from model to view space.
u_MMatrix	Matrix4	ModelMatrix4Enabled	Candera::Node	Computes the ModelMatrix from the Node's transformation data, used to transform 4-component vectors from model to world space.
u_MVMatrix	Matrix4	ModelViewMatrix4Enabled	Candera::Node, Candera::Camera	Computes the ModelViewMatrix4 from the Node's transformation data and the currently rendered Camera's ViewMatrix. Used to transform 4-component vectors from model to view space.
u_VMatrix	Matrix4	ViewMatrix4Enabled	Candera::Camera	Passes the ViewMatrix4 from the currently rendered camera to the shader. Used to transform 4-component vectors from world space to view space.
u_PMatrix	Matrix4	ProjectionMatrix4Enabled	Candera::Camera	Passes the ProjectionMatrix4 from the active camera to the shader.
u_VPMatrix	Matrix4	ViewProjectionMatrix4Enabled	Candera::Camera	Passes the active camera’s the ViewProjectionMatrix4 (View × Projection) to the shader.
u_MVPMatrix	Matrix4	ModelViewProjectionMatrix4Enabled	Candera::Node, Candera::Camera	Computes the ModelViewProjectionMatrix4 from the Node's transformation data and the currently rendered Camera's ViewProjectionMatrix. Used to transform 4-component vectors from model to homogeneous screen space.

u_CamDirection	Vector3	CameraLookAtVectorEnabled, LightActivationEnabled	Candera::Camera, Candera::Light	Passes the currently rendered Camera's look-at vector in world space to the shader. Transforms it to object space if LightsCoordinateSpace is set to Object. Can be done explicitly, or inside lighting calculations.
u_CamPosition	Vector3	CameraPositionEnabled	Candera::Camera	Passes the currently rendered Camera's position in world space to the shader.
u_Size	Float	-	Candera::PointSprite	Passes the size of a Candera::PointSprite node to the shader. This is always done, if node is type of Candera::PointSprite.
u_Material.ambient	Vector4	MaterialActivationEnabled	Candera::Material	Passes the ambient color component of the node's material (member of Appearance, see the section on material.)
u_Material.diffuse	Vector4	MaterialActivationEnabled	Candera::Material	Passes the diffuse color component of the node's material (member of Appearance, see the section on material.)
u_Material.emissive	Vector4	MaterialActivationEnabled	Candera::Material	Passes the emissive color component of the node's material (member of Appearance, see the section on material.)
u_Material.specular	Vector4	MaterialActivationEnabled	Candera::Material	Passes the specular color component of the node's material (member of Appearance, see the section on material.)
u_Material.shininess	Float	MaterialActivationEnabled	Candera::Material	Passes the shininess of the node's material (member of Appearance, see the section on material.)
u_Texture[i]	Integer	TextureActivationEnabled	Candera::Texture, Candera::BitmapTextureImage, Candera::ProxyTextureImage	Passes the activated textures handle to the shader if the textures Candera::TextureImage is of type Candera::BitmapTextureImage or Candera::ProxyTextureImage(with TextureTargetType Texture2D). i can be from 1 to 7 or nothing (e.g u_Texture, u_Texture1,..).

u_CubeMapTexture[i]	Integer	TextureActivationEnabled	Candera::Texture, Candera::CubeMapTextureImage, Candera::ProxyTextureImage	Passes the activated textures handle to the shader if the textures Candera::TextureImage is of type Candera::CubeMapTextureImage or Candera::ProxyTextureImage(with TextureTargetType TextureCubeMap). i can be from 1 to 7 or nothing (e.g u_CubeMapTexture, u_CubeMapTexture1,..).
u_Light[i].type	Integer	LightActivationEnabled	Candera::Light	Passes an integer defining the type of light i to the shader, i can range from 0 to 7.
u_Light[i].ambient	Vector4	LightActivationEnabled	Candera::Light	Passes the ambient color component of light i to the shader, i can range from 0 to 7.
u_Light[i].diffuse	Vector4	LightActivationEnabled	Candera::Light	Passes the diffuse color component of light i to the shader, i can range from 0 to 7.
u_Light[i].intensity	float	LightActivationEnabled	Candera::Light	Passes the intensity factor applied to the diffuse color of light i to the shader, i can range from 0 to 7. Ignored if the light type is Ambient.
u_Light[i].specular	Vector4	LightActivationEnabled	Candera::Light	Passes the specular color component of light i to the shader, i can range from 0 to 7.
u_Light[i].position	Vector4	LightActivationEnabled	Candera::Light	Passes the position of a point or spotlight i to the shader, i can range from 0 to 7.
u_Light[i].direction	Vector3	LightActivationEnabled	Candera::Light	Passes the direction vector of a directional or spotlight i to the shader, i can range from 0 to 7.
u_Light[i].halfplane	Vector3	LightActivationEnabled	Candera::Light	Passes the halfplane vector of a directional, point or spotlight i to the shader, i can range from 0 to 7.
u_Light[i].attenuation	Vector4	LightActivationEnabled	Candera::Light	Passes the attenuation weights of a point or spotlight i to the shader, i can range from 0 to 7.
u_Light[i].spotCosCutoff	Float	LightActivationEnabled	Candera::Light	Passes the cut off angle of a spotlight i to the shader, i can range from 0 to 7.
u_Light[i].spotExponent	Float	LightActivationEnabled	Candera::Light	Passes the spot exponent of a directional, point or spotlight i to the shader, i can range from 0 to 7.
u_Light[i].range	Float	LightActivationEnabled	Candera::Light	Passes the range of a point or spotlight i to the shader, i can range from 0 to 7.

u_Light[i].enabled	Bool	LightActivationEnabled	Candera::Light	Passes whether light i is enabled or not to the shader, i can range from 0 to 7.
u_Light[i].cameraLookAtVector	Vector3	LightActivationEnabled	Candera::Light, Candera::Camera	Computes the look-at vector of the active Camera, and passes it to the shader. Depending on the Light::CoordinateSpace the look-at vector is either in world space or in the object space of the light.
u_MorphWeight	Float Array	-	Candera::MorphingMesh	Passes the weights of a Candera::MorphingMesh to the shader. This is always done, if node is type of Candera::MorphingMesh.
u_CanvasPivot	Vector2	CanvasActivationEnabled	Candera::CanvasRenderable	Passes the position of the pivot of a Candera::CanvasRenderable to the shader.
u_CanvasSize	Vector2	CanvasActivationEnabled	Candera::CanvasRenderable	Passes the actual size of a Candera::CanvasRenderable to the shader.
ub_Material	Uniform Buffer Object	MaterialActivationEnabled	Candera::Material	Passes all of the material parameters from above in a uniform buffer object to the shader.
ub_Lights	Uniform Buffer Object	LightActivationEnabled	Candera::Light, Candera::Camera	Passes all of the light parameters from above in a uniform buffer object to the shader.
u_JointMatrices	Matrix4	SkinningMeshActivationEnable	Candera::SkinningMesh	Passes the Inverse Bind Matrix of a Joint to the shader.

Attribute Names

Candera uses pre-defined attribute names to bind a vertex buffers attributes. When writing shader programs the following attribute names shall be used:

Attribute Name	Semantic
PositionAttribute[i]	Use this attribute for passing vertex positions (local space). i ranges from 1 to 7 or is nothing (e.g. PositionAttribute, PositionAttribute1,...).
PositionTransformedAttribute[i]	Use this attribute for passing transformed vertex positions. i ranges from 1 to 7 or is nothing (e.g. PositionTransformedAttribute, PositionTransformedAttribute1,...).
NormalAttribute[i]	Use this attribute for passing the vertices normals. i ranges from 1 to 7 or is nothing (e.g. NormalAttribute, NormalAttribute1,...).

TextureCoordinateAttribute[i]	Use this attribute for passing the vertices texture coordinates. i ranges from 1 to 7 or is nothing (e.g. TextureCoordinateAttribute, TextureCoordinateAttribute1,...).
ColorAttribute[i]	Use this attribute for passing the vertices colors. i ranges from 1 to 7 or is nothing (e.g. ColorAttribute, ColorAttribute1,...).
BlendWeightAttribute[i]	Use this attribute for passing the vertices blend weight attributes. i ranges from 1 to 7 or is nothing (e.g. BlendWeightAttribute, BlendWeightAttribute1,...).
BlendIndexAttribute[i]	Use this attribute for passing the vertices blend index attributes. i ranges from 1 to 7 or is nothing (e.g. BlendIndexAttribute, BlendIndexAttribute1,...).
PointSizeAttribute[i]	Use this attribute for passing the vertices point sizes. i ranges from 1 to 7 or is nothing (e.g. PositionAttribute, PositionAttribute1,...).
TangentAttribute[i]	Use this attribute for passing the vertices tangents. i ranges from 1 to 7 or is nothing (e.g. TangentAttribute, TangentAttribute1,...).
BiNormalAttribute[i]	Use this attribute for passing the vertices binormals. i ranges from 1 to 7 or is nothing (e.g. BiNormalAttribute, BiNormalAttribute1,...).
TesselationFactorAttribute[i]	Use this attribute for passing the vertices tesselation factors. i ranges from 1 to 7 or is nothing (e.g. TesselationFactorAttribute, TesselationFactorAttribute1,...).
FogAttribute[i]	Use this attribute for passing the vertices fog attributes. i ranges from 1 to 7 or is nothing (e.g. FogAttribute, FogAttribute1,...).
DepthAttribute[i]	Use this attribute for passing the vertices depth attributes. i ranges from 1 to 7 or is nothing (e.g. DepthAttribute, DepthAttribute1,...).
SampleAttribute[i]	Use this attribute for passing the vertices sample attributes. i ranges from 1 to 7 or is nothing (e.g. SampleAttribute, SampleAttribute1,...).
CustomAttribute[i]	Use this attribute for passing your custom data per vertex. i ranges from 1 to 7 or is nothing (e.g. CustomAttribute, CustomAttribute1,...).

Single Pass Effects

Single pass effects can be achieved in one camera pass only. They are simply realized with a single Candera::Appearance attached to a Candera::Node.

Examples for single pass effects can be found in the Special Render Techniques chapter.

Local Multi Pass Effects

Local multi pass effects describe effects with local impact only, thus, they belong to one certain node. They can be realized using Candera::MultiPassAppearance. These appearances are processed in a sequence; for each appearance the node is rendered once and the results are blended together using the different RenderMode settings.

Examples for local multi pass effects can be found in the Special Render Techniques chapter.

Global Multi Pass Effects

Global multi pass (also known as screen-based effects) typically operate on render targets, often they are also referred to as post-processing effects. To realize post-processing effects first images have to be rendered into an offscreen render target. How this is done is presented in Render Targets Tutorial.

The resulting image can further be processed in any desired way.

Examples for global multi pass effects can be found in chapter Special Render Techniques.

Creating Full Screen Effects

You can of course process the resulting texture on any mesh or object you like, but often post processing effects are applied to full screen images. For that purpose a dedicated scene with only a camera and a billboard can be used. The trick is to give the Billboard a size of 1.0 unit in both dimensions.

```
Billboard* fullScreenBB = Billboard::Create(1.0F, 1.0F);
```

This causes the billboard to have a vertex buffer with coordinates that match the corners of the window/screen in Canderas viewport space. To draw the billboard full screen you just need to use a vertex shader that doesn't transform the vertex coordinates, use ReferenceShaders\Core\RefViewportSpace.vertp for this purpose. The fragment shader then does the actual post processing. As texture for the billboard you need to use the image provided by the offscreen render target.

RefViewportSpace.vertp takes two float uniforms (u_offsetLeft, u_offsetTop) as inputs, which determine the top left corner of the billboard to render. These values are defined as being in normalized viewport space, where values range from 0.0 to 1.0, where 0.0 is interpreted as the left border respectively top border of the viewport, and 1.0 as the right, respectively bottom border.

The vertex position attribute of a vertex buffer used with this vertex shader is not transformed using a model-view-projection matrix, but simply translated into OpenGL's clip coordinate space with the specified offset applied.

The following examples demonstrate how to use the RefViewportSpace.vert shader in conjunction with the depth-of-field global multi pass effect.

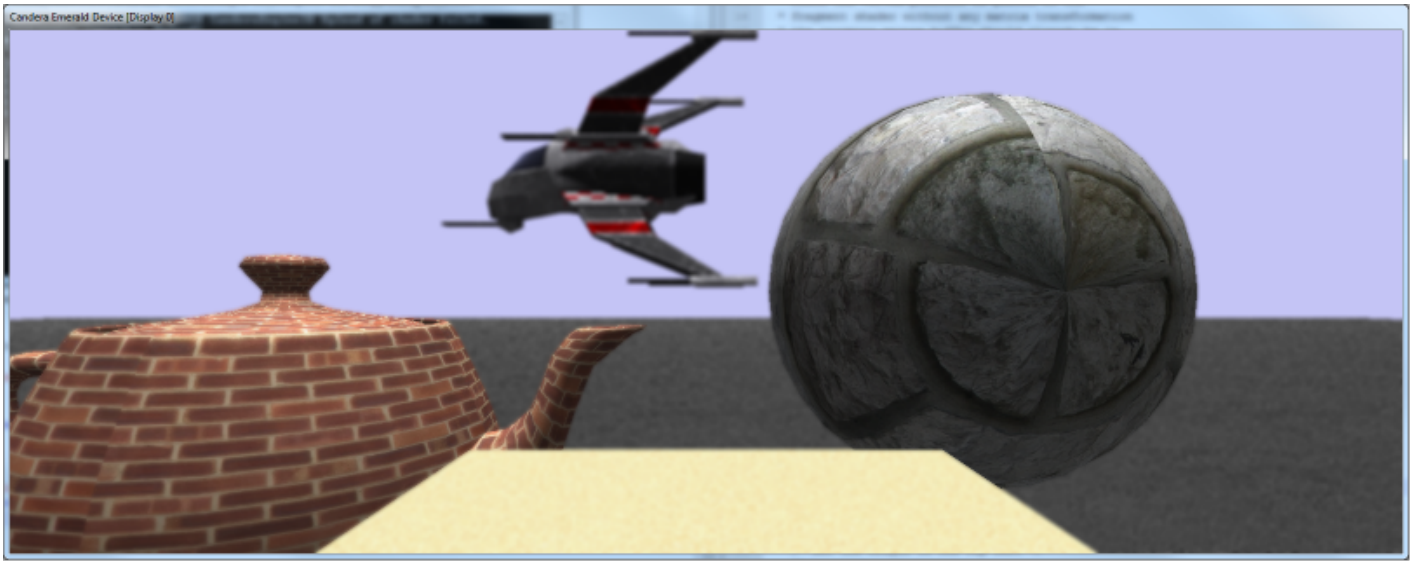
Specify a full screen rectangle:

```
//Do Shader File I/O here and initialize m_dofCombineShader shader object.
Float m_viewportLeft = 1.0F;
Float m_vieportTop = 1.0F;

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

m_fullScreenQuadDof = Billboard::Create(1.0f,1.0f);
m_fullScreenQuadDof->GetAppearance()->SetShader(m_dofCombineShader);
m_fullScreenQuadDof->GetAppearance()->SetShaderParamSetter(m_dofCombineSps);
```

This results in the following image:



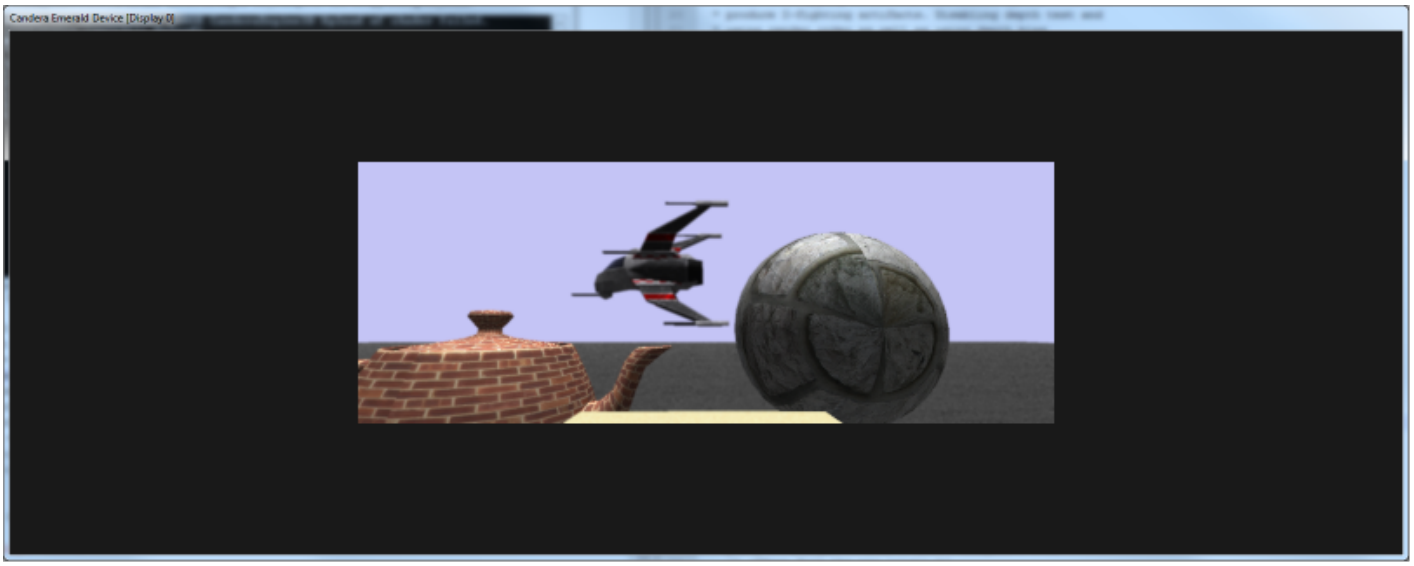
Specify a centered rectangle with half viewport resolution:

```
//Do Shader File I/O here and initialize m_dofCombineShader shader object.
Float m_viewportLeft = 0.25F;
Float m_vieportTop = 0.25F;

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

m_fullScreenQuadDof = Billboard::Create(0.5f,0.5f);
m_fullScreenQuadDof->GetAppearance()->SetShader(m_dofCombineShader);
m_fullScreenQuadDof->GetAppearance()->SetShaderParamSetter(m_dofCombineSps);
```

This results in the following image:



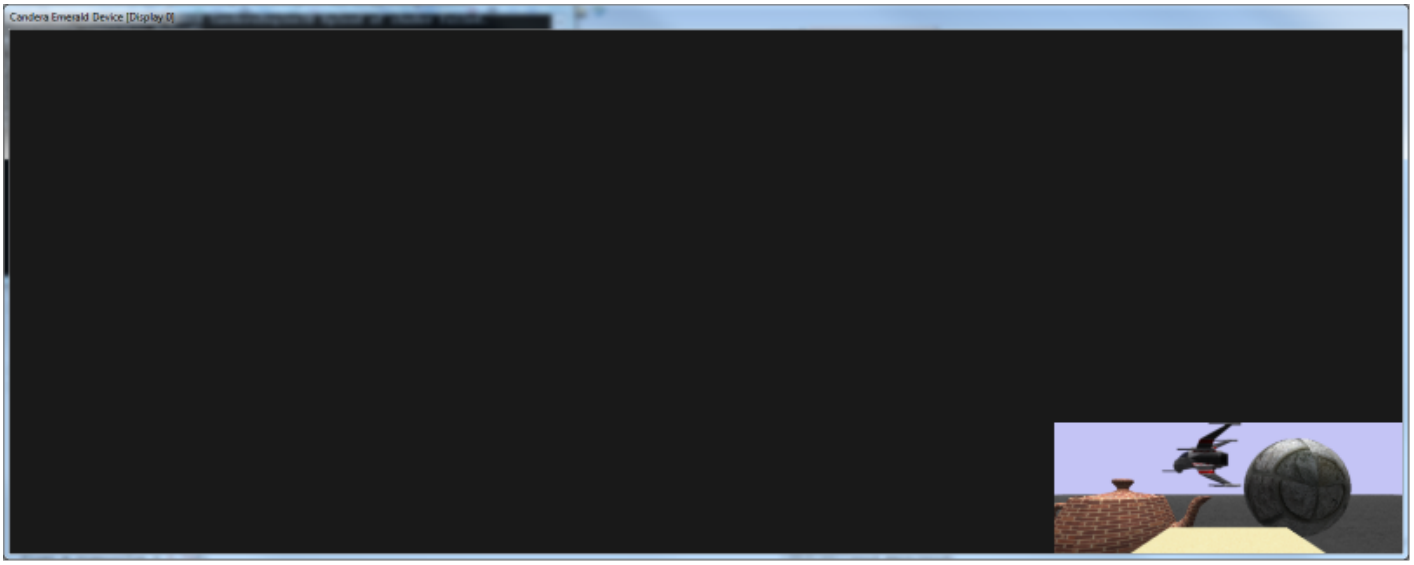
Specify a rectangle in bottom right corner with quarter viewport resolution:

```
//Do Shader File I/O here and initialize m_dofCombineShader shader object.
Float m_viewportLeft = 0.75F;
Float m_vieportTop = 0.75F;

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

m_fullScreenQuadDof = Billboard::Create(0.25f,0.25f);
m_fullScreenQuadDof->GetAppearance()->SetShader(m_dofCombineShader);
m_fullScreenQuadDof->GetAppearance()->SetShaderParamSetter(m_dofCombineSps);
```

This results in the following image:



> Overlapping Billboards (in screen space) would produce depth-fighting artifacts. Disabling depth test and using render order as well as using depth bias can be used to avoid them.

```
m_fullScreenQuadDof->GetAppearance()->SetRenderMode(RenderMode::Create());
m_fullScreenQuadDof->GetAppearance()->GetRenderMode()->SetDepthBias(0.1F);
```

For details on how to use render order please refer to Tutorial 3.

Candera Reference Shaders

Description

This document describes content and usage of Candera Reference Shaders part of CGI Studio releases. Within this document the set of Candera Reference Shaders is listed with additional information how to use and combine them in 3D applications based on Candera.

Location of Reference Shaders

Candera delivers a set of reference shader implementations, which can be found at following location:

- **\cgi_studio_candera\src\Candera\Engine3D\RefShaders**

The same set of reference shaders is available in SceneComposer.

Preferred way of usage is to export shaders to a binary asset file and to load it in an application based on Candera. Shaders part of exported scenes are automatically added to the asset file and loaded by Candera.

Used Include Files

Some parts of shader code used in Candera Reference Shaders have been collected into several files in order to be reused as include files.

Following files are available and can be used as include files:

Include File	Supports	Include In	Note	Displaces
RefEnvironment.inc	Platform specific defines	All Shaders	Shall be included in each Shader as first statement	Environment.incv, Environment.incf
RefLighting.inc	Lighting specific functions, also functions for anisotropic lighting	Shaders that support lighting	Useful helper functions for vertex lighting and pixel lighting	DefaultLighting.incv

RefTexturing.inc	Texturing specific functions, e.g. Environment Mapping	Shaders that support certain texture effects	Useful helper functions especially for common texture effects	Texturing.incv
RefSkinning.inc	Skinning specific features	Shaders that support skinning	Useful helper functions for skinning	Skinning.incv

Reference Vertex Shaders

Vertex Shaders	Supports	Combine with Fragment Shaders	GenericShaderParamSetter Configuration	Note
Core/RefTrans	Transformation, Texture	Core/RefTex, Core/RefUniColor, Core/RefUniColorTex, Core/RefTex2LightMap, Core/RefUniDiffuseMatTex, Effects/RefTexDofBlur, Effects/RefTexDofCombine	SetModelViewProjectionMatrix4Enabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true)	Lighting has no influence.
Core/RefTransColor	Transformation, VertexColor, Texture	Core/RefColor, Core/RefColorTex, Core/RefColorTexAlpha, Effects/RefColorTexAlphaOutline	SetModelViewProjectionMatrix4Enabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true)	Lighting has no influence.
Core/RefTransLight1	Transformation, Vertex Lighting with one light source, Texture	Core/RefColor, Core/RefColorTex	SetModelViewProjectionMatrix4Enabled(true) SetLightActivationEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true)	If Texture is used, depends on corresponding fragment shader only.
Core/RefTransLight2	Transformation, Vertex Lighting with two light sources. Texture	Core/RefColor, Core/RefColorTex	SetModelViewProjectionMatrix4Enabled(true) SetLightActivationEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true)	If Texture is used, depends on corresponding fragment shader only.

Core/RefTransLight3	Transformation, Vertex Lighting with three light sources. Texture	Core/RefColor, Core/RefColorTex	SetModelViewProjectionMatrix4Enabled(true) SetLightActivationEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true)	If Texture is used, depends on corresponding fragment shader only.
Core/RefTransWorldLight1	Transformation, Lighting in WorldSpace with 1 Light Texture	Core/RefColor, Core/RefColorTex	SetModelViewProjectionMatrix4Enabled(true) SetLightActivationEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true) SetLightsCoordinateSpace(World)	Lighting calculations are processed in world space instead of model space (For details see function Light::SetCoordinateSpace). If Texture is used, depends on corresponding fragment shader only.
Effects/RefTransSphereMap	Transformation, Sphere Texture Mapping	Core/RefTex, Core/RefUniColorTex, Core/RefUniDiffuseMatTex	SetModelViewMatrix4Enabled(true) SetNormalModelViewMatrix3Enabled(true) SetModelViewProjectionMatrix4Enabled(true) SetLightActivationEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true)	Used to process a sphere texture for environment mapping.
Effects/RefTransLight1SphereMap	Transformation, Vertex Lighting with one light source, Sphere Texture Mapping	Core/RefColorTex	SetModelViewMatrix4Enabled(true) SetNormalModelViewMatrix3Enabled(true) SetModelViewProjectionMatrix4Enabled(true) SetLightActivationEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true)	Used to process a sphere texture for environment mapping.

Effects/RefTransLight2SphereMap	Transformation, Vertex Lighting with two light sources, Sphere Texture Mapping	Core/RefColorTex	SetModelViewMatrix4Enabled(true) SetNormalModelViewMatrix3Enabled(true) SetModelViewProjectionMatrix4Enabled(true) SetLightActivationEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true)	Used to process a sphere texture for environment mapping.
Effects/RefTransLight3SphereMap	Transformation, Vertex Lighting with three light sources, Sphere Texture Mapping	Core/RefColorTex	SetModelViewMatrix4Enabled(true) SetNormalModelViewMatrix3Enabled(true) SetModelViewProjectionMatrix4Enabled(true) SetLightActivationEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true)	Used to process a sphere texture for environment mapping.
Core/RefTransPointSprite	Transformation of PointSprite	Core/RefPointSprite	SetModelViewProjectionMatrix4Enabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true)	Use with PointSprite. As usual for point sprites, lighting has no influence.
Core/RefTransLight1Morph	Transformation, VertexLighting with one lighe source, Morphing Weights	Core/RefColor, Core/RefColorTex	SetModelViewProjectionMatrix4Enabled(true) SetLightActivationEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true)	Use with MorphingMesh. If Texture is used, depends on corresponding fragment shader only.

Effects/RefTransLight1BumpMap	Transformation, Calculation of lighting vectors in tangent space.	Effects/RefLight1BumpMap	SetModelMatrix4Enabled(true) SetNormalModelMatrix3Enabled(true) SetModelViewProjectionMatrix4Enabled(true) SetCameraPositionEnabled(true) SetLightActivationEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true) SetLightsCoordinateSpace(World)	Used to prepare lighting vectors for bump map shaders.
Effects/RefTransLight1ProceduralWood	Transformation, Vertex Lighting with one light source, Passing position to fragment shader for procedural calculations.	Effects/RefProceduralWood	SetModelViewProjectionMatrix4Enabled(true) SetLightActivationEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true)	
Effects/RefTransAnisotropicLight1Strand	Transformation, passes vertex attributes in world space to fragment shader. Calculates direction of anisotropy, eye vector and light direction.	Effects/RefAnisotropicLight1StrandTex	SetModelViewProjectionMatrix4Enabled(true) SetModelMatrix4Enabled(true) SetNormalModelMatrix3Enabled(true) SetCameraPositionEnabled(true) SetLightActivationEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true) SetLightsCoordinateSpace(World)	

Effects/RefTransAnisotropicLight1	Transformation, Passes vertex attributes in world space to fragment shader.	Effects/RefAnisotropicLight1SpecularTex	SetModelMatrix4Enabled(true) SetNormalModelMatrix3Enabled(true) SetModelViewProjectionMatrix4Enabled(true) SetCameraPositionEnabled(true) SetLightActivationEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true) SetLightsCoordinateSpace(World)	
Effects/RefTransFur	Transformation, Indexed Multi pass upscaling in direction of normals, Texture	Effects/RefUniColorTexFur	SetModelViewProjectionMatrix4Enabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true)	Lighting has no influence.
Effects/RefTransCubeMapReflection	Transformation, Texturing reflection using one CubeMap texture	Core/RefCubeMapTex	SetModelMatrix4Enabled(true) SetNormalModelMatrix3Enabled(true) SetModelViewProjectionMatrix4Enabled(true) SetCameraPositionEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true)	Environment mapping reflections using a CubeMap texture. Lighting is not considered.
Effects/RefTransCubeMapRefraction	Transformaion, Texturing refraction using one cubemap texture.	Core/RefCubeMapTex	SetModelMatrix4Enabled(true) SetNormalModelMatrix3Enabled(true) SetModelViewProjectionMatrix4Enabled(true) SetCameraPositionEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true)	Environment mapping refractions using a CubeMap texture. Lighting is not considered. Refraction ratio has to be passed as additional uniform float refractionRatio.

Effects/RefTransCubeMapReflectionRefraction	Transformation, Texturing reflection and refraction using one cubemap texture.	Core/RefCubeMapTex2	SetModelMatrix4Enabled(true) SetNormalModelMatrix3Enabled(true) SetModelViewProjectionMatrix4Enabled(true) SetCameraPositionEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true)	Environment mapping refractions and reflections using a CubeMap texture. Lighting is not considered. Refraction ratio has to be passed as additional uniform float refractionRatio.
Core/RefTransCubeMap	Transformation, Texturing using one CubeMap texture, addressed directly by object space vertex coordinates.	Core/RefCubeMapTex	SetModelViewProjectionMatrix4Enabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true)	Direct addressing of CubeMap texture using object space vertex positions. Can be used for e.g. a SkyBox using a CubeMap texture.
Effects/RefTransDof	Transformation, Texturing with one texture, First render pass of depth-of-field rendering: Rendering the scene, Scaled depth value passed as varying.	Effects/RefTexDof	SetModelViewProjectionMatrix4Enabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true)	First pass of depth-of-field rendering has to be applied to every node that is rendered. The resulting image then contains the scene's depth values as alpha value, which is used for further processing. If other effects (e.g. Light, bump mapping etc.) have to be applied on the nodes, then also the corresponding shaders have to be adapted to meet the depth-of-field requirements. Additional uniform u_depthScale has to be passed to scale the depth values.

Effects/RefTransLight1Dof	Transformation, Texturing with one Texture Vertex Lighting with one light source, First render pass of depth-of-field rendering: Rendering the scene, Scaled depth value passed as varying.	Effects/RefColorTexDof	SetModelViewProjectionMatrix4Enabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true) SetLightsCoordinateSpace(Light::Object);	First pass of depth-of-field rendering has to be applied to every node that is rendered. The resulting image then contains the scene's depth values as alpha value, which is used for further processing. If other effects (e.g. bump mapping etc.) have to be applied on the nodes, then also the corresponding shaders have to be adapted to meet the depth-of-field requirements. Additional uniform u_depthScale has to be passed to scale the depth values. Supports per vertex lighting.
Effects/RefTransFragmentLightDof	Transformation, Texturing with one Texture, Normal and position for per fragment lighting, First render pass of depth-of-field rendering: Rendering the scene, Scaled depth value passed as varying.	Effects/RefTexFragmentLightDof	SetModelViewProjectionMatrix4Enabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true) SetLightsCoordinateSpace(Light::World); SetModelMatrix4Enabled(true); SetNormalModelMatrix3Enabled(true);	First pass of depth-of-field rendering has to be applied to every node that is rendered. The resulting image then contains the scene's depth values as alpha value, which is used for further processing. If other effects (e.g. bump mapping etc.) have to be applied on the nodes, then also the corresponding shaders have to be adapted to meet the depth-of-field requirements. Additional uniform u_depthScale has to be passed to scale the depth values. Supports per Fragment lighting.

Core/RefTransPos	Transformation, Texture, Vertex position to fragment shader	Core/RefFlatLight1	SetModelViewProjectionMatrix4Enabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true) SetLightActivationEnabled(true)	No light calculation, passing vertex position to fragment shader for FlatShading calculations.
Effects/RefTransLight1Gooch	Transformation, Assignment of ambient light component, Calculation of varyings needed for computation of specular component in world space. First pass of gooch-shading: Shading.	Effects/RefGoochShading	SetModelMatrix4Enabled(true) SetNormalModelMatrix3Enabled(true) SetModelViewProjectionMatrix4Enabled(true) SetCameraPositionEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true) SetLightActivationEnabled(true) SetLightsCoordinateSpace(Light::World)	This vertex shader in conjunction with RefGoochShading.frag p realizes the first pass of an implementation of Gooch's rendering technique to simulate the appearance of technical illustrations, often occuring in e.g. manuals or schematic presentations etc. (SIGGRAPH'98 paper "A Non-Photorealistic Lighting Model For Automatic Technical Illustration"). The first pass renders the object using a transition from "cool" to "warm" colors according to the lighting model. While phong shading is normally used to simulate real lighting, this lighting model aims more on preserving details away from the light source. The second pass then draws the silhoutte of the object. For this the object is upscaled and rendered with a fixed color (e.g. black). On Candera side the first and the second pass have to be rendered with different Culling states, such that first pass renders the front of the object, and the second only the back side.

Effects/RefTransScale	Transformation, Upscaling in direction of normals, texture coordinates.	Core/RefTex, Core/RefUniColor, Core/RefUniColorTex, Core/RefUniDiffuseMat, Core/RefTex2LightMap	SetModelViewProjectionMatrix4Enabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true)	Lighting has no influence. Used to upscale the vertices inside the vertex shader, using a float scale factor passed as uniform u_Scale. Is used for silhouette rendering in gooch-shading, but may be also used for other techniques.
Core/RefViewportSpace	Texture Adjusting the position of the screen aligned billboard by setting uniform u_offsetLeft and u_offsetTop.	Effects/RefTexGaussianBlurX, Effects/RefTexGaussianBlurY, Effects/RefTexDofCombine, Effects/RefBloomThreshold, Effects/RefCombineBloom, Core/RefInterlaceMask	SetModelViewProjectionMatrix4Enabled(false)SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true)	Used in subsequent post processing passes in conjuction with a screen-aligned billboard. The billboard should be of size 0.0 to 1.0 with 1.0 being a fullscreen billboard. The position of the billboard on the screen can be adjusted by setting the left and top offset in the range of 0.0 to 1.0. Passes the position and texture coordinate to the fragment shader without any matrix transformation. Therefore the incoming vertex buffer should already be in homogenous screen coordinates. Note: Overlapping Billboards (in screen space) would produce Z-fighting artifacts. Disabling depth test and using render order as well as using depth bias can be used to avoid them.

Effects/RefTransLight1Carbon	Transformation, Lighting with one light source, Texture, Line of Sight	Effects/RefCarbon	SetModelMatrix3Enabled(true) SetNormalModelMatrix3Enabled(true) SetModelViewProjectionMatrix4Enabled(true) SetLightActivationEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true) SetLightsCoordinateSpace(Light::Object)	Seamless texture utilized as carbon image
Effects/RefTransLight1FlopCarPaint	Transformation, Lighting with one light source, Line of Sight, Mixes two color components diffuse and emissive based on camera position	Effects/RefFlopCarPaint	SetModelMatrix3Enabled(true) SetNormalModelMatrix3Enabled(true) SetModelViewProjectionMatrix4Enabled(true) SetLightActivationEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(false) SetLightsCoordinateSpace(Light::Object)	Mixes two color components (diffuse and emissive) based on the camera position
Effects/RefTransLight1MetalFlakesCarPaint	Transformation, Lighting with one light source, Line of Sight, Texture	Effects/RefMetalFlakesCarPaint	SetModelMatrix3Enabled(true) SetNormalModelMatrix3Enabled(true) SetModelViewProjectionMatrix4Enabled(true) SetLightActivationEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true) SetLightsCoordinateSpace(Light::Object)	Metal flake effect with one noise texture

Effects/RefTransLight1FlopMetalFlakesCarPaint	Transformation, Lighting with one light source, Line of Sight, Texture, Mixes two color components diffuse and emissive based on camera position	Effects/RefFlopMetalFlakesCarPaint	SetModelMatrix3Enabled(true) SetNormalModelMatrix3Enabled(true) SetModelViewProjectionMatrix4Enabled(true) SetLightActivationEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true) SetLightsCoordinateSpace(Light::Object)	Metal flake effect with one noise texture Mixes two color components (diffuse and emissive) based on the camera position
Core/RefCanvasTrans	Applying scale and offset to vertices before transformation, Transformation into world space, Texturing with one texture.	Core/RefTex, Core/RefUniColor, Core/RefUniColorTex, Core/RefTex2LightMap , Core/RefUniDiffuseMatTex	SetModelViewProjectionMatrix4Enabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true) SetCanvasActivationEnabled(true)	Similar to RefTrans, but additionally applies canvas offset and size before multiplying the mvp matrix.
Core/RefCanvasTransLight1	Applying scale and offset to vertices before transformation, Transformation, Texturing with one tetxture, One light source with lighting in model space.	Core/RefColor, Core/RefColorTex	SetModelViewProjectionMatrix4Enabled(true) SetLightActivationEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true) SetCanvasActivationEnabled(true) SetLightsCoordinateSpace(Light::Object)	Similar to RefTransLight1, but additionally applies canvas offset and size before multiplying the mvp matrix.
Core/RefCanvasTransUniDiffuseMat	Applying scale and offset to vertices before transformation, Transformation into world space, Texturing with one texture, Forwarding of material diffuse color to fragment shader.	Core/RefCanvasUniDiffuseMat	SetModelViewProjectionMatrix4Enabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true) SetCanvasActivationEnabled(true)	Similar to RefTrans, but additionally applies canvas offset and size before multiplying the mvp matrix. Additionally it passes material diffuse color to fragment shader. This shader is used to achieve the same effect as a SolidColorBrush in 2D.

Effects/RefFxaa	Coordinates in normalized device space.	Effects/RefFxaa	SetModelViewProjectionMatrix4Enabled(false) SetLightActivationEnabled(false) SetMaterialActivationEnabled(false)	Applies the "Fast approximate anti-aliasing" algorithm to its input, which should be a quad using normalized device coordinates. The uniform (u_Resolution) needs to be set to the width and height of the input texture measured in texels.
Effects/Ref	Texture	Effects/RefTexGaussianBlurX, Effects/RefTexGaussianBlurY, Effects/RefTexDofCombine, Effects/RefBloomThreshold, Effects/RefCombineBloom	SetModelViewProjectionMatrix4Enabled(false) SetLightActivationEnabled(false) SetMaterialActivationEnabled(false) SetTextureActivationEnabled(true)	Passes texture coordinates and vertex position to fragment shader without applying a transformation. This is useful for post processing effects when the vertex buffer is just a quad in normalized device coordinates (-1,1).
Effects/RefTransGradientLinear	Transformation into world space, Vertex based linear gradient computations.	Effects/RefGradientLinear	SetModelViewProjectionMatrix4Enabled(true)	Vertex Shader for linear gradient computations.
Effects/RefTransGradientRadial	Transformation into world space, Passing vertex position to fragment shader.	Effects/RefGradientRadial	SetModelViewProjectionMatrix4Enabled(true)	Vertex Shader for radial gradient computations.
Core/RefTransLight1Skin	Skinning	Core/RefColor, Core/RefColorTex	SetModelViewProjectionMatrix4Enabled(true) SetLightActivationEnabled(true) SetMaterialActivationEnabled(true) SetTextureActivationEnabled(true) SetSkinningMeshActivationEnabled(true)	If Texture is used, depends on corresponding fragment shader only. And, use with SkinningMesh.

Vertex Shader for radial gradient computations.

Reference Fragment Shaders

Fragment Shaders	Supports	Combine with Vertex Shaders	GenericShaderParamSetter Configuration	Note
Core/RefColor	Color from varying	Core/RefTransLight*, Core/RefTransWorldLight*, Core/RefTransLight1Morph	See configuration of RefTransLight* or RefTransWorldLight1	Normaly used for models that support lighting but no texture. varying color (v_Color) is typically either set by lit material or by vertex color.
Core/RefTex	Texture	Core/RefTrans, Effects/RefTransSphereMap, Effects/RefTransScale	See configuration of RefTrans or RefTransSphereMap	Lighting has no influence.
Core/RefColorTex	Color from varying, Texture	Core/RefTransLight*, Core/RefTransWorldLight*, Core/RefTransColor, Core/RefTransLight1Morph, Core/RefTransLight*SphereMap	See configuration of RefTransLight* or RefTransWorldLight1	Normaly used for models that support lighting and one texture. varying color (v_Color) is normaly either set by lit material or by vertex color.
Core/RefUniColor	Color from uniform	Core/RefTrans, Effects/RefTransScale	See configuration of RefTrans	Useful if certain color is set via uniform (u_Color) directly. Lighting or vertex colors have no influence.
Core/RefUniColorTex	Color from uniform, Texture	Core/RefTrans, Effects/RefTransSphereMap, Effects/RefTransScale	See configuration of RefTrans or RefTransSphereMap	Useful if certain color is set via uniform (u_Color) directly and blended with one texture. Lighting or vertex colors have no influence.

Core/RefUniDiffuseMatTex	Material diffuse color, Texture	Core/RefTrans, Effects/RefTransSphereMap	See configuration of RefTrans or RefTransSphereMap	Final color = materials diffuse color * texel color. The uniform value of the materials diffuse color is set by Candera automatically if a Material is applied to a Node. Lighting or vertex colors have no influence.
Core/RefPointSprite	Pointsprite incl. Texturing	Core/RefTransPointSprite	See configuration of RefTransPointSprite	Used for point sprites only.
Core/RefTex2LightMap	Texture, LightMap	Core/RefTrans, Effects/RefTransScale	See configuration of RefTrans	Lightmap Multitexturing: A Lightmap typically defines an illumination texture at unit 1 that is multiplied with texture at unit 0.
Effects/RefLight1BumpMap	Per pixel lighting using the normals supplied in u_Texture1 (normal map). Computed color is combined with the texture in u_Texture0.	Effects/RefTransLight1BumpMap	See configuration of RefTransLight1BumpMap	Used to display rough surfaces without adding polygon complexity to it. The shader accepts two textures. u_Texture0 defines the color map used. u_Texture1 defines a normal map, that stores normals in tangent space to be used for lighting calculations.
Effects/RefProceduralWood	Procedural calculation of interpolated rings, which appears like annual rings of wood if the right colors are chosen.	Effects/RefTransLight1ProceduralWood	See configuration of RefTransLight1ProceduralWood	For procedural calculations two color, the orgin of the annual rings and a multiplier that controls the densitiy of the rings can be passed.
Effects/RefAnisotropicLight1StrandTex	Per pixel anisotropic lighting, using a strand texture	Effects/RefTransAnisotropicLight1Strand	First texture for color, second for strand.	The strand texture must hold the diffuse and specular light intensity in the R and G channels, respectively.
Effects/RefAnisotropicLight1SpecularTex	Per pixel anisotropic lighting	Effects/RefTransAnisotropicLight1	See configuration of RefTransAnisotropicLight1	Using alternative anisotropic specular highlights, according to Ward's SIGGRAPH 92 paper "Measuring and Modeling Anisotropic Reflection". Additional calculated specular component is combined with u_Texture0. Used to display the reflection behavior of multiple surfaces. Uniforms u_AlphaX and u_AlphaY model the distribution of the specular highlight. Combined with a texture it can for example be used to simulate metallic surfaces like brushed steel. In the cited paper alphaX and alphaY for several materials can be found.

Effects/RefUniColorTexFur	Assignment of product of alpha mask, texture and fixed color.	Effects/RefTransFur	See configuration of RefTransFur	Simulates the appearance of fur if applied with MulitPassRendering. Is also intended to demonstrate multi pass rendering.
Core/RefCubeMapTex	Addressing one cube map texture using a single varying vec3.	Effects/RefTransCubeMapReflection, Effects/RefTransCubeMapRefraction, Core/RefTransCubeMap	See configuration of RefTransCubeMapReflection, RefTransCubeMapRefraction, RefTransCubeMap	Cube Map addressing, material colors or lighting are not considered.
Core/RefCubeMapTex2	Mixing two different addressed fragments of a CubeMap using a varying ratio. Addressing is done using two varying vec3.	Effects/RefTransCubeMapReflectionRefraction	See configuration of RefTransCubeMapReflectionRefraction	Cube Map addressing, material colors or lighting are not considered.
Core/RefUniDiffuseMat	Material diffuse color	Core/RefTrans	See configuration of RefTrans	Final color = materials diffuse color The uniform value of the materials diffuse color is set by Candera automatically if a Material is applied to a Node. Lighting or vertex colors have no influence.
Effects/RefTexDof	Assignment of Texture, First render pass of depth-of-field rendering: Rendering the scene, Depth encoded into alpha channel.	Effects/RefTransDof	See configuration of RefTransDof	First pass of depth-of-field rendering has to be applied to every node that is rendered. The resulting image then contains the scene's depth values as alpha value, which is used for further processing. If other effects (e.g. Light, bump mapping etc.) have to be applied on the nodes, then also the corresponding shaders have to be adapted to meet the depth-of-field requirements.
Effects/RefColorTexDof	Color from varying, Assignment of Texture, First render pass of depth-of-field rendering: Rendering the scene, Depth encoded into alpha channel.	Effects/RefTransLight1Dof	See configuration of RefTransLight1Dof	First pass of depth-of-field rendering has to be applied to every node that is rendered. The resulting image then contains the scene's depth values as alpha value, which is used for further processing. If other effects (e.g. bump mapping etc.) have to be applied on the nodes, then also the corresponding shaders have to be adapted to meet the depth-of-field requirements. Support per vertex lighting.

Effects/RefTexFragmentLightDof	Per fragment lighting with one light source, Assignment of Texture, First render pass of depth-of-field rendering: Rendering the scene, Depth encoded into alpha channel.	Effects/RefTransFragmentLightDof	See configuration of RefTransFragmentLightDof	First pass of depth-of-field rendering has to be applied to every node that is rendered. The resulting image then contains the scene's depth values as alpha value, which is used for further processing. If other effects (e.g. bump mapping etc.) have to be applied on the nodes, then also the corresponding shaders have to be adapted to meet the depth-of-field requirements. Supports per fragment lighting.
Effects/RefTexDofBlur	Assignment of texture, second render pass of depth-of-field: blurring the texture, simulating an optical circle of confusion, caused by lens refraction out of focus.	Core/RefTrans	See configuration of RefTrans	Second pass of depth-of-field rendering: blurring the incoming image. Result is a blurred image of the passed texture. u_CocScale controls the size of the circle of confusion, and therefore the blur strength.
Effects/RefTexDofCombine	Third pass of depth of field: Assignment and blending of two textures, representing the sharp and blurred image. Blend weights result from focus and focus range of the simulated camera lens.	Core/RefViewportSpace, Effects/Ref	See configuration of RefTrans	Third pass of depth rendering: Combines the sharp and the blurred image according to the set focus of the simulated camera lens. u_Focus controls the distance of the focus plane (where a sharp image would be formed). u_Range controls the size of the area around the focus plane, where a sharp image would be drawn.
Effects/RefTexGaussianBlurX	Assignment of texture, Subsequent pass of post-processing effect: blurring the texture with a Gaussian blur, Gaussian blur step width from unifrom	Core/RefViewportSpace, Effects/Ref	See configuration of RefViewportSpace plus SetUniform("u_blurFactor",Shader::Float, &blurFactor)	Subsequent pass of post-processing effect: blurring the incoming image horizontally. Result is a blurred image of the passed texture. u_blurFactor controls the step width of the Gaussian function. u_blurFactor should always be 1.0/TextureWidth so that every step will correlate with one pixel.

Effects/RefTexGaussianBlurY	Assignment of texture, Subsequent pass of post-processing effect: blurring the texture with a Gaussian blur, Gaussian blur step width from unifrom	Core/RefViewportSpace, Effects/Ref	See configuration of RefViewportSpace plus SetUniform("u_blurFactor",Shader::Float, &blurFactor)	Subsequent pass of post-processing effect: blurring the incoming image vertically. Result is a blurred image of the passed texture. u_blurFactor controls the step width of the Gaussian function. u_blurFactor should always be 1.0/TextureHeight so that every step will correlate with one pixel.
Core/RefFlatLight1	Per fragment lighting with one light source for Flat Shading.	Core/RefTransPos	See configuration of RefTransPos	In order to use derivative functions dFdx and dFdy the extension GL_OES_standard_derivatives has to be enabled and supported by driver.
Effects/RefGoochShading	Per Fragment calculation of specular component, Mixing cool and warm color using the dot product of light direction and vertex normal as weight, Output: Mixed color + specular and ambient lightcomponent.	Effects/RefTransLight1Gooch	See configuration of RefTransLight1Gooch	This fragment shader in conjunction with RefTransLight1Gooch.vertp realizes the first pass of an implementation of Gooch's rendering technique to simulate the appearance of technical illustrations, often occuring in e.g. manuals or schematic presentations etc. (SIGGRAPH'98 paper "A Non-Photorealistic Lighting Model For Automatic Technical Illustration"). For details please refer to the description of the vertex shader, and to the cited paper. The following unifiorms are needed: u_CoolColor and u_CoolColor are 4-component vectors that describe the colors for the cool to warm transition. u_CoolDiffuseWeight and u_WarmDiffuseWeight are float values that describe, how the original diffuse color is combined with the warm and cool colors.
Effects/RefBloomThreshold	Assignment of texture, Subsequent pass of bloom effect: extracting only the bright parts of the image that should be blurred, brightness threshold from unifrom	Core/RefViewportSpace, Effects/Ref	See configuration of Core/RefViewportSpace	Subsequent pass of post-processing bloom effect: extracting only the bright parts of the incoming texture. Those bright parts are blurred in the next step to generate a bloom effect.

Effects/RefCombineBloom	Subsequent pass of bloom effect: Assignment and djustable combination of the original and the bloomed texture	Core/RefViewportSpace, Effects/Ref	See configuration of RefViewportSpace	Subsequent pass of post-processing bloom effect: The incoming texture of the original scene and the incoming texture of the bright, bloomed parts are combined. The blending of the two textures is adjustable by uniforms representing the intensity and saturation of the images.
Core/RefInterlaceMask	Uses a texture's color to create a mask through discarding pixels with low transparency. No value is written to the discarded fragments, so also no depth or stencil values.	Core/RefViewportSpace	See configuration of RefViewportSpace	Is especially useful for e.g. Creating stencil masks for stereoscopic 3D.
Effects/RefCarbon	Wraps a seamless texture multible times around the object to achieve carbon look.	Effects/RefTransLight1Carbon	See configuration RefTransLight1Carbon	Creates a car paint with typical carbon look.
Effects/RefFlopCarPaint	Mixes the two color components together based on the camera position and a user defined bias	Effects/RefTransLight1FlopCarPaint	See configuration RefTransLight1FlopCarPaint	Creates a car paint with shiny, anisotropic two color look.
Effects/RefMetalFlakesCarPaint	Adding a noise to the spectral part of the light. Noise can be wrapped around the object multible times. Create the illusion of metal flakes in the car paint.	Effects/RefTransLight1MetalFlakesCarPaint	See configuration RefTransLight1MetalFlakesCarPaint	Creates a car paint with shiny, sparkling look produced by metal flakes within specular highlight

Effects/RefFlopMetalFlakesCarPaint	Mixes the two color components together based on the camera position and a user defined bias. Adding a noise to the spectral part of the light. Noise can be wrapped around the object multible times. Create the illusion of metal flakes in the car paint.	Effects/RefTransLight1FlopMetalFlakesCarPaint	See configuration RefTransLight1FlopMetalFlakesCarPaint	Creates a car paint with shiny, sparkling look produced by metal flakes within specular highlight combined with anisotropic two color look.
Core/RefCanvasUniDiffuseMat	Assignment of material's diffuse color to fragment color	Core/RefCanvasTransUniDiffuseMat	See configuration RefCanvasTransUniDiffuseMat	This shader is used to achieve the same effect as a SolidColorBrush in 2D.
Effects/RefFxaa	Coordinates in normalized device space.	Effects/RefFxaa	See configuration of Effects/RefFxaa	Applies the "Fast approximate anti-aliasing" algorithm to its input, which should be a quad using normalized device coordinates. The uniform (u_Resolution) needs to be set to the width and height of the input texture measured in texels.
Effects/RefGradientLinear	Mixing fragment color according to gradient computations.	Effects/RefTransGradientLinear	See configuration RefTransGradientLinear	Fragment Shader for linear gradient computations.
Effects/RefGradientRadial	Assignment of radial gradient color to fragment color.	Effects/RefTransGradientRadial	See configuration RefTransGradientRadial	Fragment Shader for radial gradient computations.
Core/RefColorTexAlpha,	Assignment of product of varying color and texture color to fragment color.	Core/RefTransColor	See configuration RefTransColor	This fragment shader shall be used for text or alpha mask rendering.

Effects/RefColorTexAlphaOutline	This fragment shader renders an outline of color "u_outlineColor" and width specified by "u_outlineWidth" around the contents in the texture "u_Texture". The input texture is also multiplied by the vertex color. If the "u_outlineWidth" value is bigger than 10, the contents of the texture need be bigger than 10 too, otherwise artefacts will start to appear with increasing outline width. There is no instancing version of this shader, because it will usually be applied to text rendering which is batched by Canvas rather than the Renderer.	Core/RefTransColor	See configuration RefTransColor Additionally: SetTexelSizeActivationEnabled(true) Custom uniforms: uniform vec4 u_outlineColor; // Color of outline. uniform float u_outlineWidth; //Width of outline.	This fragment shader shall be used if a text or an alpha mask shall receive an outline.