Candera V3.0.2

Release Date: February 19, 2015

To keep backwards compatibility for any interface change in the public API the old interface has been kept and marked as deprecated. The deprecated interfaces will be removed in future versions. Compiler warnings will be generated when using a deprecated interface through the usage of Candera Macro CANDERA_DEPRECATED_3_2_0.

Candera Engine Base

Resource Objects.

Large data buffers that need to be uploaded or that are rarely used are not longer directly stored in DeviceObject objects. Instead, a ResourceDataHandle structure is stored, which can consist of one of :

- pointer to existing data, pointer to disposer function and size (backward compatible)
- information on where data can be found in asset and size
- custom information for custom ResourceProviders.

ResourceObjects are objects that guarantee that during their lifetime, data referenced by a ResourceDataHandle is available.

Bitmap Resource Objects.

The pixel buffer is stored in the new ResourceDataHandle structure. See migration guide for an example on how it can be used.

Introduce C++ keyword override in Candera

In a method declaration, override specifies that the function must be overriding a base class method.

The keyword override is part of C++ 11 standard and supported by MSVC 11 and gcc 4.7. It helps to prevent unintentional newly created functions, e.g. due to typos, that are intended to override virtual base function.

Candera Engine 3D

Shader: Support for precompiled shaders on all enabled platforms.

Uploading shader programs using Shader::Upload can now be done in a more flexible way. Until now it was only possible to upload either shader source or pre-compiled binaries, through a target-specific implementation of RenderDevice::CompileAndLoadShaderPair.

Now it is possible to upload (and compile if necessary) either shader source or shader binary objects. It is even possible to upload complete shader programs in order to avoid shader linking. How shader upload is done depends on the kind of data that is available in the asset and the capabilities of the target platform, however at runtime this is done completely transparent to the user.

Advantage is that the more flexible way of runtime created shader source code (as used in Candera2D implementations over OpenGL ES) is supported as well as using pre-compiled shaders to save execution time for compiling them. While using pre-compiled shaders on enabled platform is recommended due to performance reasons, still backward compatibility to existing solutions using uncompiled shaders is given.

OpenGL ES 2 Reference Shaders relocated

The Candera OpenGL ES 2.0 Shader language reference Shaders have been moved from `<cgi_studio_candera>/src/Candera/Engine3D/ReferenceShaders` to `<cgi_studio_candera>/src/Candera/Engine3D/RefShaders/ESSL1`

OpenGL ES 3 Reference Shaders

Candera now contains a set of reference Shaders for the OpenGL ES 3.0 Shader language. `<cgi_studio_candera>/src/Candera/Engine3D/RefShaders/ESSL3`

Shader Resource Objects.

Fragment and vertex shader data or access information is stored in the new ResourceDataHandle structure. See migration guide for an example on how it can be used.

VertexGeometry Resource Objects.

The vertex, index and element format arrays are stored in the new ResourceDataHandle structure. See migration guide for an example on how it can be used.

VertexBuffer primitive iteration.

Iteration over triangles is replaced by primitive iteration using a forward iterator, which is more performant, allows iteration over point and line primitives and supports multiple parallel iterations. See migration guide for an example on how it can be used.

VertexBuffer vertex access.

Vertex access interface is moved from the VertexBuffer to a separate class for better performance in case of vertex data accessed from assets. See migration guide for an example on how it can be used.

8/16/32 bit indexed array buffers.

VertexGeometry accepts index buffers that consist of 8, 16 or 32 bit indexes. The configuration of index size can be set through the new values of BufferType enumeration: UInt8IndexedArrayBuffer, UInt16IndexedArrayBuffer and UInt32IndexedArrayBuffer.

VertexElementFormat access.

VertexElementFormat access interface is moved from the VertexGeometry to a separate class for better performance in case of vertex data accessed from assets. See migration guide for an example on how it can be used.

VertexGeometryBuilder support for 8/16/32 bit index buffers.

VertexGeometryBuilder can create index buffers with values on 8, 16 or 32 bit, depending on the number of vetices, or on the configured format if explicitly set.

BlendEquation Min/Max support.

Min/Max options were added to the RenderMode BlendEquation, which specifies how a new pixel is combined with previously rendered pixels in the frame buffer.

Candera Engine 2D

Node2D Listener support.

Node2DListener defines hooks that are called when certain Node functions are triggered, e.g. when a Node's transformation changes.

Auto Arrangement for GridLayouter

Two new methods available for GridLayouter:

- GridLayouter::SetGridAutoArrangement : sets the grid autoArrangement. See method description for possible layouter arrangement settings.
- GridLayouter::GetGridAutoArrangement: retrieves the grid autoArrangement settings of a given node, set by GridLayouter::SetGridAutoArrangement method.

Candera System

Different namespace than 'Candera' for EnumDataTypes

```
#define ENUM_DATA_TYPE_NAMESPACE MyNamespace        // <-- new!

#define ENUM_DATA_TYPE \
    ENUM_DATA_TYPE_BEGIN(SomeEnum) \
        ENUM_DATA_TYPE_ITEM(SomeItem0) \
        ENUM_DATA_TYPE_ITEM_VALUE(SomeItem1, 4) \
        ENUM_DATA_TYPE_ITEM(SomeItem2) \
        ENUM_DATA_TYPE_ITEM_INVISIBLE(InvisibleItem) \
        ENUM_DATA_TYPE_ITEM_VALUE(SomeItem3, 7) \
    ENUM_DATA_TYPE_END(SomeEnum)

#include <Candera/System/MetaInfo/EnumDataType.h>
```

If ENUM_DATA_TYPE_NAMESPACE is defined before including EnumDataType.h, the enum 'SomeEnum' will be placed into MyNamespace. Without that (ENUM_DATA_TYPE_NAMESPACE is not defined), SomeEnum is placed automatically into namespace Candera.

Candera Device

Uniform Location Caching

Introduced Uniform Caching for Auto uniforms to increase performance. Calls to GetUniformLocation now pass semantic instead of names as parameter, which speeds up comparison within cache.

Maximum Element Index capability

RenderDeviceCapabilities structure contains a new field for the maximum value of vertex index that is supported by the device.

RenderDevice::DrawVertexBuffer

RenderDevice::DrawVertexBuffer handles drawing of a VertexBuffer, regadless of its configuration (primitive types, index formats, upload location).

Added formats on DirectTextureImage

Additional formats have been added to DirectTextureImage::Format, unpacked RGBA and BGRA texture format.

Shader Upload functions now support Upload of precompiled binary shaders.

The function RenderDevice::CompileAndLoadShaderPair only supports the fixed Upload of text or binary shader objects. In order to provide a more flexible handling of shader Upload, this function has been split up into RenderDevice::UploadShaderSource, RenderDevice::UploadShaderBinary and RenderDevice::UploadShaderBinaryProgram. Additionally the function RenderDevice::UploadShader has been added which calls the appropriate shader upload function transparently to the user. This interface enables the user to choose whether shader shall be available as binary or source shader objects.

Candera AssetLoader

Resource Objects.

The font and raw resources are stored in the new ResourceDataHandle structure. See migration guide for an example on how it can be used.

Dispose after upload.

The dispose after upload feature of DefaultAssetProvider has become deprecated with the introduction of resource objects. The data is no longer attached to the device objects, but rather it is loaded from the asset upon necessity.

TextEngine

Glyph Spacing

Glyph Spacing is available on TextBrush, and LayoutingOptions. This allows a user to correct the spacing between glyphs, by adding a specified number of pixels to the spaces between glyphs.

BitmapFont Engine

BitmapFont Engine is available for usage inside Candera-based applications. A bitmap font is essentially an image file which consists of several characters images and a header that depicts the size and location of each character inside the image. Each bitmap font might contain several fonts - mainly one font face with multiple sizes. One advantage of utilizing bitmap fonts is that the rendering to the screen requires very little resources and since each character is represented as a sub-texture, they can be reused without requiring additional memory on the GPU.

In order to use BitmapFonts, the CMake variable **CANDERA_FONTENGINE** should be changed to *BitmapFont*.

> Harbuzz Text Shaper cannot be used with BitmapFont, either ComplexScriptLib or NoShaping should be chosen.

This font engine uses a proprietary format to describe glyph bitmaps and glyph meta info. In order to use a bitmap font, provide an implementation for Candera::TextRendering::FontStore.

Bitmap fonts require memory mapped assets. So, when the faces are loaded in be sure the storageType is set correctly:

```
TextRendering::FontResourceDescriptor
descriptor.storageType = TextRendering::FontResourceDescriptor::MemoryResource;
```

Generating Bitmap Fonts

The following steps need to be considered:

- Install Bitmap Font Generator from http://www.angelcode.com/products/bmfont/

Open the application. From the Options menu:

- Choose Font Settings. Set as needed the settings for "Font", "Add font file", "Size" etc.
- Configure the Export Options. Set "Font descriptor" to "XML" and Textures to "png - Portable Network Graphics"
- "Save bitmap font as ..."

Convert the resulted XML and PNG files to binary format using the internal tool BmFontCreator using the following command:

- BmFontCreator. exe -o MyFont32.fbm *.fnt

> A bitmap font should be used at its native pixel size. However, they are scalable because of the use of bitmaps, but only scaling down is recommended. In case of up scaling, the visual quality will be reduced. Including separate font sizes for bitmap fonts is possible in order to achieve the best looking results.

Glyph Substitution

A substitution character may be set within the style with the interface Style::SetDefaultCodepoint. This character is used by the text engine in any of the following situations:

- when a glyph is missing from the style. i.e. no matching font can provide that glyph. Exception make control character, like newline or LRE.
- when UTF8 buffers contain malformed characters.

Character Glyph Mapping

GlyphBitmap now contains information about the position within the text from where the glyphs originated, and the location of the glyph within the font.

Glyph Order

The order in which glyphs are generated may be controlled by setting a property within the TextRenderContext. See TextRenderContext::GlyphOrder for details. This is most relevant when doing preprocessing. OriginalGlyphOrder is intended for use when intermediate measurements of the text are required, for instance when truncating texts. OrigitnalChunkOrder is more performant

then RenderOrder, and is intende for measurements, where the order of the chunk is not important, for instance instance when computing layout rectangles. RenderOrder shall be used when rendering.

Text Preprocessing

The text engine allows text to be preprocessed by providing utility the class PreprocessingContext. This interface is satisfied by two concrete instances.

- MinimalPreprocessingContext - stores all the information need to be able to render the glyphs.
- MaximalPreprocessingContext - stores all the information available within the text renderer for each glyph. TextBrush allows setting PreprocessedText instead of Text, to speed up text processing within the brush.

Migration Guide

Following an overview about interface changes between Candera V3.0.1 and Candera V3.0.2.

Description	Candera V3.0.1	Candera V3.0.2
Shader Resource Object	const void* fragmentShader = shader->GetFragmentShader(); const void* vertexShader = shader->GetVertexShader();	Shader::ShaderResource fragmentShaderResource(shader->GetFragmentShaderResourceHandle()); const void* fragmentShader = fragmentShaderResource.GetData(); Shader::ShaderResource vertexShaderResource(shader->GetVertexShaderResourceHandle()); const void* vertexShader = vertexShaderResource.GetData(); NOTE: data is guaranteed only until ShaderResource objects are destroyed by going out of scope.
Vertex Geometry Resource Object	bool isGeometryMutable = vg->IsMutable(); const void* vertexArray = vg->GetVertexArray(); void* vertexArray = vg->GetVertexArray(); const UINT16* indexArray = vg->GetIndexBuffer(); UINT16* indexArray = vg->GetIndexBuffer(); const VertexGeometry::VertexElementFormat* formatArray = vg->GetVertexElementFormat(); VertexGeometry::VertexElementFormat* formatArray = vg->GetVertexElementFormat();	bool isVertexArrayMutable = vg->GetVertexArrayResourceHandle().m_isMutable; VertexGeometry::VertexArrayResource vertexArrayResource(vg->GetVertexArrayResourceHandle()); const void* vertexArray = vertexArrayResource.GetData(); void* vertexArray = vertexArrayResource.GetMutableData(); bool isIndexArrayMutable = vg->GetIndexArrayResourceHandle().m_isMutable; VertexGeometry::IndexArrayResource indexArrayResource(vg->GetIndexArrayResourceHandle()); const void* indexArray = indexArrayResource.GetData(); void* indexArray = indexArrayResource.GetMutableData(); bool isFormatArrayMutable = vg->GetFormatArrayResourceHandle().m_isMutable; VertexGeometry::FormatArrayResource formatArrayResource(vg->GetFormatArrayResourceHandle()); const VertexGeometry::VertexElementFormat* formatArray = formatArrayResource.GetData(); VertexGeometry::VertexElementFormat* formatArray = formatArrayResource.GetMutableData(); NOTE: data is guaranteed only until resource objects are destroyed by going out of scope.
Bitmap Resource Object	bool isMutable = bitmap->IsMutable(); const void* pixels = bitmap->GetPixels(); void* pixels = bitmap->GetPixels();	bool isMutable = bitmap->GetPixelsResourceHandle().m_isMutable; Bitmap::PixelsResource pixlesResource(bitmap->GetPixelsResourceHandle()); const void* pixels = pixlesResource.GetData(); void* pixels = pixlesResource.GetMutableData(); NOTE: data is guaranteed only until resource objects are destroyed by going out of scope.

Font/Raw Resource Object	ResourceHandle fontHandle = provider->OpenFontResourceById(fontId); if (fontHandle != 0) { UInt32 fontSize = provider->GetResourceSize(fontHandle); const void* fontAdress = provider->GetResourceAddress(fontHandle); provider->ReadResource(fontHandle, fontData, 0, fontSize); provider->CloseResource(fontHandle); }	ResourceDataHandle fontHandle = provider->GetFontResourceById(fontId); UInt32 fontSize = fontHandle.m_size; const void* fontAddress = ResourceData::GetAddress(fontHandle); ResourceData::CopyData(fontData, fontHandle, 0, fontSize); NOTE: data is guaranteed only until resource objects are destroyed by going out of scope.
VertexBuffer Primitive Iteration	VertexBuffer* vb; const VertexBuffer::Triangle* triangle = vb->GetFirstTriangle(); while (triangle != 0) { UInt16 x = triangle->index[0]; UInt16 y = triangle->index[1]; UInt16 z = triangle->index[2]; triangle = vb->GetNextTriangle(); }	VertexBuffer* vb; for (VertexBuffer::PrimitiveIterator it = vb->GetPrimitiveIterator(); it.IsValid(); ++it) { const VertexBuffer::Primitive& primitive = *it; UInt16 x = primitive.m_index[0]; UInt16 y = primitive.m_index[1]; UInt16 z = primitive.m_index[2]; }
VertexBuffer Vertex access	VertexBuffer* vb; const void* vertex = vb->GetVertex(0); void* mutableVertex = vb->GetMutableVertex(0);	VertexAccessor accessor(VertexAccessor(*vb->GetVertexGeometry())); const void* vertex = accessor.GetVertex(0); void* mutableVertex = accessor.GetMutableVertex(0);
VertexElementFormat access	VertexGeometry* vg; bool isPosition = vg->DoesVertexUsageExistAtUsageIndex(VertexGeometry::Position, 0); const VertexGeometry::VertexElementFormat* positionElementFormat = vg->GetVertexElementFormatAtUsageAndUsageIndex(VertexGeometry::Position, 0);	VertexGeometry* vg; ElementFormatAccessor accessor = ElementFormatAccessor(*vg); bool isPosition = (accessor.GetElementFormatByUsageAndIndex(VertexGeometry::Position, 0) != 0); const VertexGeometry::VertexElementFormat* positionElementFormat = accessor.GetElementFormatByUsageAndIndex(VertexGeometry::Position, 0);

Shader Upload	const void* fragmentShader = shader->GetFragmentShader(); const void* vertexShader = shader->GetVertexShader(); Handle fragmentShaderHandle; Handle vertexShaderHandle; CompileAndLoadShaderPair(vertexShader, vertexShaderHandle, fragmentShader, fragmentShaderHandle);	const void* fragmentShader = shader->GetFragmentShader(); const void* vertexShader = shader->GetVertexShader(); Handle fragmentShaderHandle; Handle vertexShaderHandle; Shader::BuildType buildType; GetShaderBuildType(vertexShader, buildType); if (buildType == Shader::ShaderSource) { UploadShaderSource(Shader::VertexShader, vertexShader, vertexShaderHandle); } else { ... } GetShaderBuildType(fragmentShader, buildType); if (buildType == Shader::ShaderSource) { UploadShaderSource(Shader::FragmentShader, fragmentShader, fragmentShaderHandle); } else { ... } Note: This is normally done transparently by Candera.

CGI-Studio CMake Configuration

- The CMake macro `CgiCreateWidgetSet` now creates an empty WidgetSet if no additional widgets have been added.
- `-m32` has been added as default compiler flag for gcc builds to ensure compilation even on x64 gcc.
- With update of Attribute Caching the user variable CANDERA_MAX_SHADER_ATTRIBUTE_COUNT became obsolete and has been deleted.