Projektpraktikum aus Computergraphik und Digitaler Bildverarbeitung (WS2010)
Franz Spitaler, Przemyslaw Musialski (Editors)

Documentation for the 'Advanced Facade Rendering' Project

Franz Spitaler †

21 February 2011

Abstract

This document is intended to give an overview of the project 'Advanced Facade Rendering'. The Project's aim was to improve the generation of facades and their geometry and subdivisions based on simple facade pictures. These subdivisions are set automatically by using some sophisticated image-based algorithms or manually by the user. This way it is possible to create astonishing results of a facade from a picture in just a few minutes.

1. Introduction

The aim of this document is to give an overview of the project and to explain how everything works in general. We start explaining the overall project structure by explaining the library - dependencies and afterwards we continue with a more detailed description of the rendering part of the project. We will describe how the rendering is organized and will go through the processing steps needed to calculate one frame. A list of all classes and a short description of their functionality finishes the documentation part of this document. At last a small conclusion follows and gives a hint on how the framework could (and will) be extended. The project was programmed for the course 'Praktikum aus Computergraphik und digitaler Bildverarbeitung' which has 12ECTS, and i worked together with Dr. Przemyslaw Musialski who was my advisor for this course. Before we start with the descriptions of the libraries and their dependencies, we want to give you a first impression of how the program looks like when you use it. In figure 1 you see the renderer part of the program.

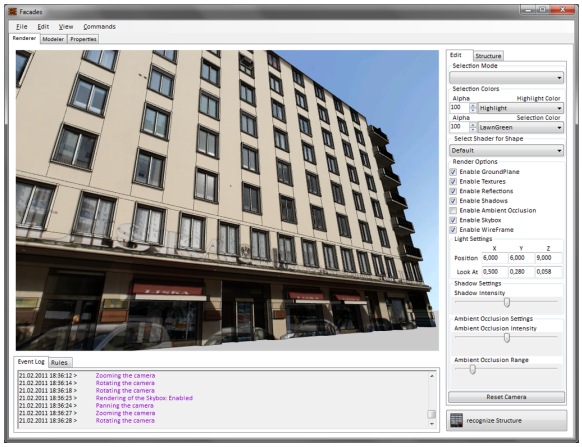

Figure 1: *This is the GUI of the program. You can see the logger's RichTextBox at the bottom, the many render settings on the right side and the menu bar. In the middle region you see the OpenGL control and an edited and rendered facade.*

2. Program structure

Our goal while working on the project, was to keep the structure simple. So it is much easier to understand what is being processed in which part of the program. Therefore we decided to create several libraries that all work together. These libraries are set up in a way, such that tasks and computations dedicated to one topic, are handled and processed in one library without the need to interact with other libraries. All in all we created eight libraries that somehow depend on each other. We also tried to decouple them as much as possible in the time we had. A short introduction to the structure of the program follows right after this section (see section 2.1 and following).

2.1. Program subdivision

The project is split in two parts to guarantee a good maintainability. You can see the resulting main-structure in the figure 2 below. This architecture allows us to keep a good overview

† spitefr@gmail.com, Mat.#:0226436

of the project and it was easy to change or add features to the program.

- The first part of the program holds only one library. It's name is 'Facade Renderer' and it handles every possible user input. The main use of the library is the creation of the user interface (the window) and the right handling of the interactions with the user. It delegates the different inputs and settings-changes to the right destination and is the place where every part of the program interacts with other parts.
- The second part of the program consists of seven libraries that form the base for all computations. Most of the libraries are used in the 'user-interface'-part of the application. They receive commands from the user-interface and act in a defined manner. A description of the libraries follows in the next section 2.2 and in section 4 on page 6.

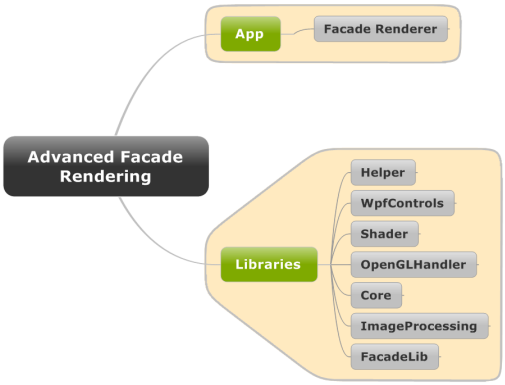

Figure 2: *The two main parts of the project visualized.*

2.2. Library description

When we defined the architecture of the program we made the decision to split the user interface part of the program from the rest of the program, which is a common program design. We created seven libraries that are used for handling the different parts of the program.

- The smallest and least complex library is the **'Helper'**. It has no dependencies on other libraries of the project and can be used throughout the whole program, because it is referenced in every other solution project. The helper's only task is to log messages to a RichTextBox in a 'Winforms' window.
- The second project is the **'WpfControl'** which is used for displaying and changing properties of WPF-elements. It also consists only of one class.
- The **'Shader'** project is the first project that is little more complex and consists of a few classes that handle the interaction and use of OpenGL-shaders.
- The **'OpenGLHandler'** is the main rendering project. Is is responsible of the rendering of the scene and uses for example the shader project to create the rendered images.
- The **'Core'** is mainly used as an extension to the OpenGLHandler and is responsible for the texture setups and the materials used to render the images. It is also a bridge from the OpenGLHandler project to the FacadeLib project because it defines some important interfaces and primitives that are also used in the FacadeLib.
- The **'ImageProcessing'** library: It implements all needed algorithms needed to do image based calculations like edge detection. These algorithms are able calculate positions of facade-splits for example. This is probably the most complex library in the whole solution.
- The **'FacadeLib'** library is also relatively complex. It defines a lot of interfaces and data types that are used to represent the facade (both, in the modeler and the renderer parts of the solution). Most of the user-interactions are delegated to and processed in this project. It is the main place of the definitions of the 'facade' and it's elements.
- The last library is the **'Facade Renderer'** which defines all the user interface parts of the program for example. It responds to user inputs directly (e.g. by handling the 'Open File' command) or delegates commands to the right destination in another project (eg. by calling the 'RenderFrame' method defined in the OpenGLHandler library).

2.3. Library dependencies

After the short introduction into the base structure of the solution we want to discuss the inter-library dependencies. To show how they all form the solution we decided to create the figure 3. This is a visualization of the dependencies between the solution projects. The arrow indicates a dependency. If project 'A' depends on project 'B', an arrow will be pointing from the box with the label 'A' to the box with the label 'B'.

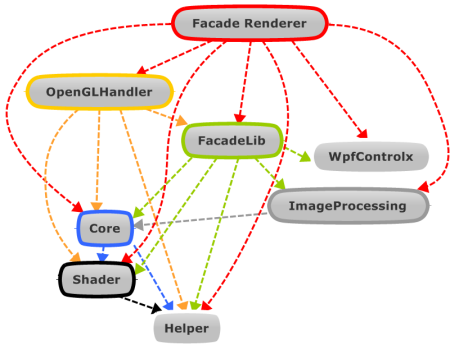

Figure 3: *Visualization of the dependencies between the libraries of the solution.*

3. The rendering

After this short descriptions about how the whole solution is organized and set up, we continue with a very important part of the project - the rendering part. At first there was the need to make the decision which OpenGL-wrapper for C# we wanted to use to create the visual output of our program. We decided to use the OpenTK toolkit because it is a very easy to use library. A short description of the OpenTK library follows in the next section (see 3.1)

3.1. The OpenTK library

The OpenTK library is a good wrapper for the OpenGL library. It makes all OpenGL commands and objects usable in the C# language. It defines a very useful 'glControl' winforms form which is used in out solution.
There are many useful parts in OpenTK and a few were used in out project. There are the Vector and Matrix classes for example which we used in out solution. These classes are very useful when interacting with OpenGL objects or shaders. The classes define all standard operators and provide geometric functions like 'transform' or 'scale' as well.
Another very nice feature when using the built-in vector and matrix-classes was the fact, that these classes are used by default to send data to the shaders, so no additional conversion of the data was necessary.

3.2. The Core library

The core library is used as the central point of handling the use and the creation and generation of OpenGL textures that we use to generate the output images. It additionally provides the interface and the classes of the different materials that we used to render the facade. There are the 'MaterialBase' and the 'ReflectMaterial' classes. The third purpose of the library is the definitions of the primitives used throughout the program. Theses are the 'Ray', the 'Vertex' and the 'Box' classes for example.

3.2.1. Textures

We need to use several different textures in out program. They are of a different type, like 'Texture2D' or 'TextureCubeMap' for example and they use different color formats like 'Rgba' or 'DepthComponent24'. We need to load textures from the file-system or create them from 'Bitmap' objects. We also need to create empty textures with some predefined settings. These type of textures are used to render information of the scene to (like the depth-information that is used to compute the shadows in the scene). This creation and loading of the textures is done in the 'Texture' class and is used in the 'Texturehandler'. This two classes work together very tightly and the texturehandler uses the functions defined in the texture class. It also created all framebuffer objects needed in the program. FBO's are used to render the depth informations of the scene directly into a texture for example. This texture is then used to calculate the shadow information for the final image. Another use of the FBO's is the use in the SSAO calculation.

3.2.2. Materials

Since we want to render facades we need to use two different types of materials at least. These materials are the 'MaterialBase' material and the 'ReflectMaterial'. The first one is used for all facade elements by default. The second material is derived from the first one and adds a reflection parameter. It is used for windows in the facade for example. Both materials implement the 'IMaterial' interface that is defined also in the core library. This interface defines properties that every material should have, the correspondent shader program and shader-interaction functions. The materials are used when we render the facade elements. They provide automatic shader-setup so it is easy to work with them.

3.2.3. Primitives

The core library also defines some important primitive classes that we use. There is the 'Ray' class for example. This class is used to determine the actually pointed-at facade element for example. The 'Vertex' is used to store informations of one single point of the facade. This information includes the position, the normal and the texture coordinates that should be used to texture the shapes. This vertex is used by several other primitive type in the core library. These are the 'edge', the 'triangle' and the 'quad' classes especially. Each of these classes use the vertex class to define it's endpoints. Another use of the vertex is the definition of the points of a 'ShapeGL' object which effectively is a cuboid. The primitives defined here in the core library are used in several different other libraries. So it is possible for different libraries to use the same objects to work with and therefore it is easily possible for them to interact with other libraries.

3.3. The OpenGLHandler library

The OpenGLHandler is the library where all OpenGL rendering functions and setups are defined and handled. The most important class in this library is the static 'Render' class which in the central place for all OpenGL specific operations. At first we will discuss the initializations steps that are done in the render class (see section 3.3.1 followed by a small introduction of the other important classes of this library (see sections 3.3.2, 3.3.3 and section 3.3.4) and their use in the render class. At last we will show how the different renderpasses are done (see section 3.3.5) and how the results of each pass is used in the frame composition (see section 3.3.6).

3.3.1. Initialization

The initialization of the OpenGL part of the program is done after all 'glControls' that we use are initialized and a handle

is created for them. All the initialization steps are invoked from the 'glControl_HandleCreated' method in the 'MainForm' class.

- The first thing is the initialization and setup of OpenGL itself in the function 'Render.InizializeRenderer'. In that function all important and constant OpenGL settings are specified. This function also initializes all shaders that we use in out program and after all of the shaders are compiled and added to the shadermanager the few additional scene objects like the light, the groundplane and the 'screenalignedquad' (used to render the skybox) are set up. When all these steps are done, the first step of initializations is finished. At this point we can use OpenGL calls and have a set up scene with shaders ready for use.
- The next step in the initialization process is the setup of the camera for the glControl. The setup for the camera is simply done by setting the viewport size in the camera and by setting a middle vector for it. These informations are needed for the rendering itself and also for the user interactions like the rotation of the camera around a point in space.
- The last step in the chain of initializations is the creation of all default textures and all framebuffer objects. This initialization is done in the 'TextureHandler.Initialize' function.

3.3.2. Light

The light class is used to automatically handle settings that affect the light. This includes the 'direction' of the light in the scene. The light we use in the program is a infinitely distant light. This means that we only need the direction of the light to render. Since it is more easy for the user do define a light's position and a light 'look at position' it is possible to define the direction of the light by setting these two values. The light class also provides automatic calculation of the modelmatrix and projection matrix that are used to render the scene into a depthbuffer as seen from the light's point of view. This depthbuffer is then used to calculate shadow positions in the scene when we re-render everything from the camera's point of view.
Since our lightsource should behave, like it was infinitely far away from the scene, we create an orthogonal projection matrix. When we create the projection matrix there is one special optimization that we use to achieve a high accuracy for the depthmap calculation. We calculate the minimum and the maximum distance of the boundingbox of the whole facade. We then use these values to set the near and far values of the projection matrix resulting in depthvalues that always lie in the range [0, 1].
When we do the distance calculation we also calculate the minimum and maximum x- and y- values of the boundingbox points as seen from the camera. With these points we can improve the shadowmap quality again because the shadowmap is perfectly fit onto the facade.

3.3.3. Camera

The camera is an important part of the rendering library. We defined the way the camera should work by letting it rotate around a defined point in space if the user changes the rotation. This way we ensure that the facade is in the center of the view normally. Of course it is possible not only to rotate the camera, but also to move and zoom it. We define the center of the rotation of the camera in the center of the facade object. The camera object itself is responsible for the handling of the user inputs that changes the actual view. So it provides functions that calculate translations, zooms and rotations from user inputs.
Every time there is a change of the view, we additionally calculate a new projection matrix. We perfectly fit the facade's boundingbox into the depth range by setting the near and far projection values accordingly. This ensures a really good 'depth resolution' which is needed when we calculate the SSAO. The only problem that occurs when we set the depth range to fit the facade's boundingbox is, that also the groundplane would be culled at these depth-values. To solve this problem we use another projection matrix that has a very small value for the near and a very high value for the far settings. The camera has another feature built-in. Is is possible to go into a 'focus-mode'. This means that we can select facade elements and let the camera zoom in, to fit the selections on the screen. This is done by calculating the intersection of all the boundingboxes of the selected facade elements. Then this boundingbox will be fit on the screen (with a margin). This focus-mode also sets another point of rotation for the camera, namely the center of the boundingbox of the selected facade elements. This helps the user when he wants to watch the elements from all points of view.
When we switch from the normal mode into the focus mode (by pressing 'f' while we have selected at least one facade element) we will see another nice feature that we implemented in this library, the 'Animate' class. An introduction to this class will be given in the next section.

3.3.4. Animate

As mentioned above, we use a short animation to show the state-change from normal render mode to the focus mode that includes a camera position change and a transparency change of the non-selected facade elements that are blended out. The animation values are calculated in the animate class that is also located in the OpenGLHandler library. The usage of this class is very simple. It uses a delegate that can be set to different predefined functions to get the animation the user wants. A whole animation is then started (after setting the delegate!) by calling the 'Animate.RenderSequence' function. This function uses the previously set delegate to change the desired values and then calls the render method. These steps are done until the animation is finished.

3.3.5. Renderpasses

In our program we need to use more than one render pass to get the final output. We want to give a short overview of all used render passes that we need to get the results we want. Not all render passes are needed to be computed every time a new frame should be output. For example the scene is only rendered to the depthmap of the light if changes of the light of changes of the facade require the depthmap to be updated. If only the camera is rotated for example, there is no need to update the shadowmap. Similarly the SSAO texture is only rendered if there is the need to do so (i.e. when SSAO is enabled in the render settings). So here is a list with with short descriptions of all render passes that can occur.

- **Depthmap pass:** In this render pass we render the facade from the light's point of view. We use backface-culling (with a polygon-offset) to render into the shadowmap to avoid light-leaking. As mentioned earlier, this pass is not required every frame. We only render to the shadowmap if the settings of the light changed or if at least one facade element was modified.
- **Main render pass:** This pass is rendered from the camera's point of view. This render pass is in fact one of two passes that are required for every frame. This render pass's result is the normally shaded scene. If the shadow mapping is enabled we use the shadowmap to compute the shadows as seen from the camera. If the shadowmapping is disabled we will not do this computation. The results of this pass are in fact three different textures that can be used in further passes. The first texture holds the color information including alpha values. The second texture holds the depth information of the rendered scene as seen from the camera's point of view. The third texture holds the normals for each fragment in the camera-view-space. This additional information is needed for the SSAO pass.
- **Screen space ambient occlusion pass:** This render pass is used to calculate the ambient occlusion information from the previously created depth texture and normal texture from the main render pass. Together with a 'random texture' is is possible to calculate an approximations of the ambient occlusion as seen from the camera. In this render pass we do not actually render geometry but do some calculations based on the textures mentioned that are mapped to a screen aligned quad (in fact is is a triangle fan, because quads are not supported in OpenGL 3.2 any more). The result of this renderpass is another texture that is going to be blurred in the next render pass. This pass as well as the blurring pass are only done if the ambient occlusion is enabled in the render settings.
- **Blurring pass:** This pass's purpose is only to blur the resulting texture from the ambient occlusion pass and it will only be executed if the ambient occlusion is enabled. To blur the ambient occlusion texture from the previous pass we additionally use the depth texture from the main render pass. This texture is used for the bilateral blur because it prevents blurring over a big 'depth discontinuity', so no bleeding from the blur will occur.
- **Texture combination pass:** This pass is the second pass that will be executed every time although it is not required if the ambient occlusion if disabled. It is used to multiply the color texture from the main render pass by the blurred ambient occlusion texture from the ambient occlusion passes. If the ambient occlusion is disabled the color texture will be multiplied by a white texture instead of the ambient occlusion texture.

In the next section you will see how the render passes work together.

3.3.6. Frame composition

To give an overview of the different render passes and to illustrate how they work together and form the final render result we want to refer to figure 4. The green boxes represent the different render passes and the gray boxes their output textures. The sequence of the render passes from top to bottom correspond to the time they are calculated relatively compared to the others. The output of each render pass is at least one texture the output of the last render pass is not really a texture but directly rendered to the screen. The orange arrows between the previously calculated textures to the following render passes indicate their usage in these render passes.

3.3.7. Effects

To create the desired effects of the final output of the rendering part we use a few special effects as already mentioned. We want to give a short description of how they work. We will start with the one's that were easy to implement and finish with the more complex one's.

3.3.7.1. Reflections The first effect is the calculation of reflections in the windows of the facade. As already mentioned we use two different materials to give the windows and other elements of the facade a different appearance (see section 3.2.2). To achieve a more realistic rendering we decided to implement a reflection effect for facade elements that are defined to use the reflection shader. This shader uses the direction from the camera to the vertices of the shapes to calculate the reflections. This is achieved by reflecting this vector with the normal information on the surface. The result is another vector which is then used to lookup the texture color from the cubemap that is also used to render the skybox. This way it is possible to see the correct reflection of the skybox in every window.

3.3.7.2. Shape Wireframes Since we work with many facade elements (i.e. shapes) we need to distinguish every element from it's neighbors. When we only use the shading information this is often difficult, so we wanted a better solution and came up with the wireframe mode. To get the

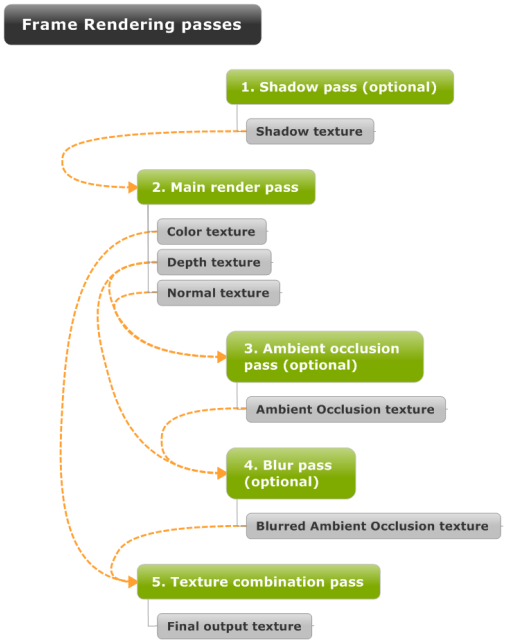

Figure 4: *Visualization of the different render passes and how the individual resulting textures are used in the other render passes.*

desired result we render every 'real' edge of the facade elements before we rerender the elements shaded again. This was only possible by adding a second vertexbuffer object o the ShapeGL's because the sequence of the rendered vertices is completely different than the one that is used to render the facade element shaded normally.

3.3.7.3. Shadow Mapping This is a very common effect and widely used nowadays. To add shadows in our program we decided to implement a simple standard shadowmapping and use a percentage closer filtering to get smooth transitions from 'shadowed' to 'non-shadowed'. We render the scene from the light's point of view whenever it is necessary (see sections 3.3.2, 3.3.5 and figure 4 for more information). The calculated depthmap is used in the next render pass - the main render pass (again, see 4).

3.3.7.4. Screen space ambient occlusion If the ambient occlusion effect is enabled, we calculate the effect in the optional render passes four and five (see figure 4). To do that we create a small (32x32 pixel in out solution) random texture that hold randomly generated values. We also need the depth texture and the normal texture created in the main render pass. The random texture is used to 'randomly' rotate 16 predefined vectors in the shader that is used to calculate the ambient occlusion information. For every pixel the view-space position is calculated and then translated by the rotated vectors. Then a depth comparison with the depth texture is performed and this way we calculate an occlusion value for that fragment. The result is an approximation of the ambient occlusion effect by only using depth- and normal- information. The result is also noisy because we always use different vectors to calculate the ambient occlusion value. This is the reason why we use a blur pass to smooth out the noise of the calculated textures.

4. A more detailed description of the libraries

We already gave a short introduction to the different libraries we use (see section 2.2). To give an even better insight into the program we now want to describe the different libraries in detail below.

4.1. Helper

The Helper project only consists of one static class and does not actually add functionality to the program. It's intent is to help the used get response to actions and inputs he makes.

4.1.1. Logger

The Logger is a simple class that uses a RichTextBox to log some messages that should be output. The output destination for the Logger is set as soon as the handle of the RichTextBox is created. It defines an enum called 'LogType' that is used to distinguish different types of messages that should be logged. Since the output of the Logger is set at the time the handle of the RichTextBox is created, it could be possible that messages that should be logged arrive before that time. To handle such situations there is a 'buffer' that is used to store all output informations until the output-textBox is created.
It is possible to change the behavior of the logger, because it is possible to enable or disable the whole logger or only some types of log-messages. The logger does not have any dependencies on other solution libraries and is therefore usable in every other library of the solution.

4.2. WpfControl

This is a relatively simple library. It is the second library that has no dependencies on oder libraries of the solution. It defines a WPFControl that is used to display settings of different controls.

4.2.1. WpfPropertyGrid

The class defines a WPFControl that is derived from the grid control. It provides functionality to represent properties of wpf elements. These properties can then also be changed and are immediately updated on that wpf element. It is used in the modeler part of the project.

4.3. Shader

The Shader project only consists of three classes that handle all used shaders in the solution. The ShaderManager and the ShaderLoader are two public static classes and together with the Shader class this library handles every action on the shaders.

4.3.1. ShaderManager

The static ShaderManager class is used to interact with the different shader objects in a simple way. It uses a dictionary that holds all loaded shaders, so that they can be accessed easily using their names. The manager allows us to add shaders, set shaders active and get information about the different shader objects (e.g. the shaderprogram handle).

4.3.2. Shader

The Shader class is used to handle all interactions with a single shader. It can load the sourcecode of a shader (using the ShaderLoader), compile, link and completely create a single OpenGL shader that is added to the shadermanager's shader dictionary after successful creation. It stores all data needed for a shader like the sourcecode, the shader handles and the shader-program handle.

4.3.3. ShaderLoader

The static ShaderLoader class is used to load single shaders sourcecodes. It is only used by the Shader class and retrieves the sourcecodes for a specified shader from the resources of the solution project not the filesystem (this could be changed easily and is the reason why we use it at all).

4.4. OpenGLHandler

This project is used to interact with OpenGL. This could be the rendering of a frame or the animation of the camera object. For a detailed description of the OpenGLHandler please see section 3.3.

4.5. Core

This project is used to interact with OpenGL also. For a detailed description please see section 3.2.

4.6. ImageProcessing

This is the central library for image-based calculations. It consists of many classes and is the most complex library in our solution. It provides a lot of classes and algorithms that can be used to extract information from images. For example we have the ability to do edge detection as well as other, more complex, calculations.

4.7. FacadeLib

The facadelib library uses most of the other libraries except the OpenGLHandler library. It is a main part of the program where many informations and calculations are invoked. The facadelib defines the actual behavior and 'structure' of the facade we work with. This definitions are done in the first class we will look at, now.

4.7.1. FacadeControl

The FacadeControl is defined as a static class that get's initialized on program startup. It provides the load and save methods used to load a saved file or to save a file (it uses the ShapeSerializable to do that). The facadecontrol class also defines some other important features, like the ability to use the well known 'undo' and 'redo' commands. To be able to do that, we use the Serializer again and push or pop the actual facade on a stack. This works fast and stable. The facadecontroller has a lot of different members that interact with each other. Some important one's are the 'MainShape' which represents the root shape of out shape hierarchy. If we split this shape into two shapes these two shapes will be children of the root shape. That way we get a 'shape-tree' we can work with. Another important member of this class is the 'RenderableList' which is used to hold the 3d geometry that represents the shape tree. This is the actual geometry we use to render that facade in 3d. The last important member is the 'MaterialDict' dictionary which holds all (i.e. two) materials we use for rendering the facade elements. There are important methods in this class as well, namely the 'Selection and Interaction Functions', the 'Groups Functions' and the 'Travesal Functions'.

4.7.2. ImageManager

This is another static class we use. It mainly provides some imaging extensions like the possibility to convert a 'Bgr' object to a 'Vector3' and vice versa.

4.7.3. InteractionEvents

This is again a static class we use to define some interactions with ShapeWPF objects. Two important methods are the 'AutoSplitX' and 'AutoSplitY' methods which use the FacadeOptimizer to compute splitlines for the selected shapes.

4.7.4. InteractionManager

An object of this class is generated only in the FacadeControl and then used in a few other classes as well. It is used to interact with the WPF representations of the facade elements and splitlines and therefore in includes many 'Traversals and Updates' methods that are used in the modeler of out project.

4.7.5. ISelectable

The facadelib does not only define some 'facade classes' but also some interfaces like the ISelectable interface. This interface is used to define objects to be selectable. It defines

one property named the 'SelectionState' that indicated the state of this object. It also defines some methods that indicate if an object's boundingbox is hit by a ray or not, and it defines the method to determine the exact position of that hit. This interface is implemented by two other interfaces that we will describe in the following sections, namely the 'IEditable' and the 'IRenderable' interfaces.

4.7.6. IEditable

This is an interface that implements the ISelectable interface. It defines some properties like the 'Pos', 'Size' and 'BBox' that define the geometric properties of objects. It contains a shape object as well as the properties that indicate if this IEditable object 'IsRoot' or 'IsTerminal'. The IEditable interface is implemented in the abstract class 'EditableShape'.

4.7.7. IRenderable

This interface also implements the ISelecatble interface. It additionally implements the IDisposable interface to ensure that every implementing classes must implement the 'Dispose' method as well. This is needed to ensure that, if an object that should be rendered and therefore has some OpenGL-Objects associated, all of these OpenGL-objects must be released as well. Additionally the interface defines two properties. This is the 'Material' and the 'Opacity' properties which are needed for all IRenderable objects. The defined methods include the 'Render' method that ensures that every IRenderable is able to render itself. It provides an additional 'RenderShadow' method were we can save processing time by not setting useless states etc. and the 'RenderWireFrame' method which uses another vertex buffer object to only render the wireframe representation of an IRenderable. An 'Update' and a 'ChangeState' method finish the definitions made here. The IRenderable interface is implemented by the 'ShapeGL' class which also derives from the 'EditableShape' abstract class.

4.7.8. ISplittable

The last interface that is defined in the FacadeLib is the ISplittable interface. It implements the 'Core.IIndexable' interface that only defines a 2d integer vector 'Index'. The ISplittable interface defines two properties that are IEnumerbable<float> that represent the x- and y- values of the splitlines. Also four definitions of methods are included, they define some splitline interactions like the adding of a splitline or the adding of a splitline to it's parent, the splitline removal or the movement of a splitline. This interface is implemented by the 'Shape' class.

4.7.9. Groundplane

As mentioned in the short description of the FacadeLib itself, the library also defines the important shape objects and it's WPF and OpenGL representations. It also defines the 'Groundplane' for example which we want to describe here. It is basically a very simple object that is rendered with OpenGL. So it defines some standard members like a material, a size and position. This class implements the IDisposable interface to ensure OpenGL cleanup on object destruction as well. As the name suggests this class is used to render a groundplane beneath the facade to be able to show the shadows and to make the spatial relations of the objects clearer.

4.7.10. ScreenAlignedQuad

The ScreenAlignedQuad is also a very simple class that define objects that should be rendered to the screen. This object is even simpler than the groundplane that was discussed in the section above because we don't need a size and other informations. This object is a triangleFan to be exactly because in OpenGL 3.2 there are no quads any more. The object's vertices are always defined in normalized screen coordinates and have the values [-1; 1] for x- and y- coordinates. These screenalignedquads are used to render textures to the screen. This is needed when we do the ambient occlusion calculations for example. In the shaders we have the advantage that we do not have to transform the vertices of the object at all.

4.7.11. Shape

This is another very important class of our solution. It's purpose is to represent a facade element. So all facade elements are shapes. The shape class implements the ISplittable interface that enables objects to get split. It has several members that allow us to index that shape in different ways and it has some standard members like a size etc. It includes several state members that indicate the selection state of the objects for example and it has a IEnumerable collection that hold the children of this shape.

4.7.12. ShapeGL

The ShapeGL is the class that is used to render all of the facade elements. It provides a constructor that creates geometry from a shape object. The ShapeGL class derives from the abstract EditableShape class and implements the IRenderable interface. The class provides methods to create the actual geometry of the object and all methods needed to render the ShapeGL, it's wireframe representation and to update the geometry.

4.7.13. ShapeWPF

This class is equivalent to the ShapeGL class with the difference that it creates a WPF representation from a Shape object.

4.7.14. ShapeGroupWPF

This class is used to group ShapeWPF objects as the name might suggest. This class handles the possible interactions with groups of shapes like the 'synchronized splitting' of many shapes at once.

4.7.15. SerializableFacade, SerializableShape, SerializableGroup

These classes are used to define the serialization of the shapes, shapegroups and the whole facade. With the serialization of these objects it is possible to load and save data. This is used in combination with files ans with memory streams to provide the undo and redo commands.

4.8. FacadeRenderer

To finish the detailed description of out solution it is also very important to get an overview of the GUI-library we use in our solution. This library is used to define all GUI elements and also includes the program entry point in the static class 'Program'. The FacadeRenderer library references all other libraries described (see also figure 3 for more information on the library dependencies). The program class opens one window at startup, and this window is of type 'MainForm'. This class is the first we will discuss in the next sections.

4.8.1. MainForm

This class is the main class for the user interface and defines all winforms controls you will see on the startup of the program. There are several settings-related controls as well as the RichTextBox the logger will use (see section 4.1 and section 4.1.1 for more information on the logger) and the OpenGL control (for more information on th OpenTK library used see section 3.1). The MainForm is derived from the 'Form' class that gives us many possibilities to handle user interactions and events that occur. The class is relatively big, so we decided to split it up to several parts. In the main part of the class we handle some user inputs like the changes of some settings and the closing of the application itself. The three other parts of the class have different aims. The first other part handles all user interactions done on the menu. This is the creation or the loading of a new facade for example. The second other part of the MainForm class is dedicated to handle all events that have to do with the OpenGL control. This includes view changes as well as the handleCreated event for example. The last other part of the MainForm is used to handle events that have to do with the RichTextBox used to log messages to. It's main purpose is to set the output destination for the logger if the textbox was created handling the handleCreated event.

4.8.2. LogSettingsForm

This is another form that is used to enable the user to specify which messages should be logged into the textbox. There are several checkboxes in the form that change the behavior of the logger directly because it can be enabled or disabled completely. Another possibility is to enable or disable special types of messages.

5. Conclusion

To conclude the documentation, it's important to note, that there are many possibilities to improve the behavior and performance of the program. Some of the features we implemented were very hard to implement and therefore time consuming. Since we only had a small amount of time to finish the project we didn't have the time to optimize everything. It is also possible to add some nice features like a rule extraction from the facade elements to allow the user to interactively change the size and the look of the generated facade.