IDP // Message-based Unified Rendering of 2D and 3D Scenes

Project Documentation

Verena Dogeanu, Benedikt Brandner

Definitions

Table Screen: The screen inside the table, showing a top down perspective of the scene

Presentation Screen: The screen next to the table, showing a perspective view and allowing sketches.

Previous Work

The rendering of the previous system and information flow of the implementation state previous to our project was as follows: During normal operation, the Table Screen rendering of buildings, streets, greenspaces, etc. is done with Qt, through custom commands to QPainter. The menu and compass overlays are rendered as a mix of QPainter commands and native OpenGL commands.

When any plugin is launched, the PluginContext is filled with information about current scene objects and is rendered on top of the normal Table Screen content, but underneath the menu overlay. Most plugins clear the screen entirely and render relevant scene elements themselves. This leads to little to no possibility of executing multiple plugins in parallel. Since plugins used to render only to the Table Screen, most of them restrict themselves to 2D calculations and drawing.

The presentation screen is drawn inside a Qt Window, the 3D scene is rendered through VL[1], a lightweight abstraction on top of OpenGL. Unfortunately, VL seems to be used by very few people, and thus has little extra examples, documentation or community. In addition it has quite some unobvious quirks and specialties that are not documented.

Motivation & Goals

The situation described above leads to a number of shortcomings and possible improvements that motivated this IDP.

- Allow multiple plugins to run in parallel
- View plugin data in 3D scene
- Deduplicate rendering code of common elements, e.g. buildings, streets, …

C++ Renderpipeline

To achieve said goals, we decided to move all rendering code to the C++ side. Plugins would be able to communicate additional scene elements to render through message-passing. This switch in technology impacts all plugins, since their rendering code has to be adapted for use with the new pipeline. Thus this was also a good opportunity to define the new interface as 3D-only, making the Table Screen an orthographic top-down projection of an otherwise 3D scene, which also makes rendering plugins in 3D a trivial task. It is up to the plugins, however to present their data in a way that makes sense on the Presentation Screen.

The main challenge was thus to migrate the existing 3D rendering code from the PresentationFramework to the main project and extending it with features required for the Table View, such as different colorings, dynamic hiding of objects, etc.

[1] http://visualizationlibrary.org/docs/1.0/

Class Overview

This section presents an overview of the main Classes that constitute the new rendering pipeline and tries to describe the rough information flow. Below purposes for all new classes are described briefly.

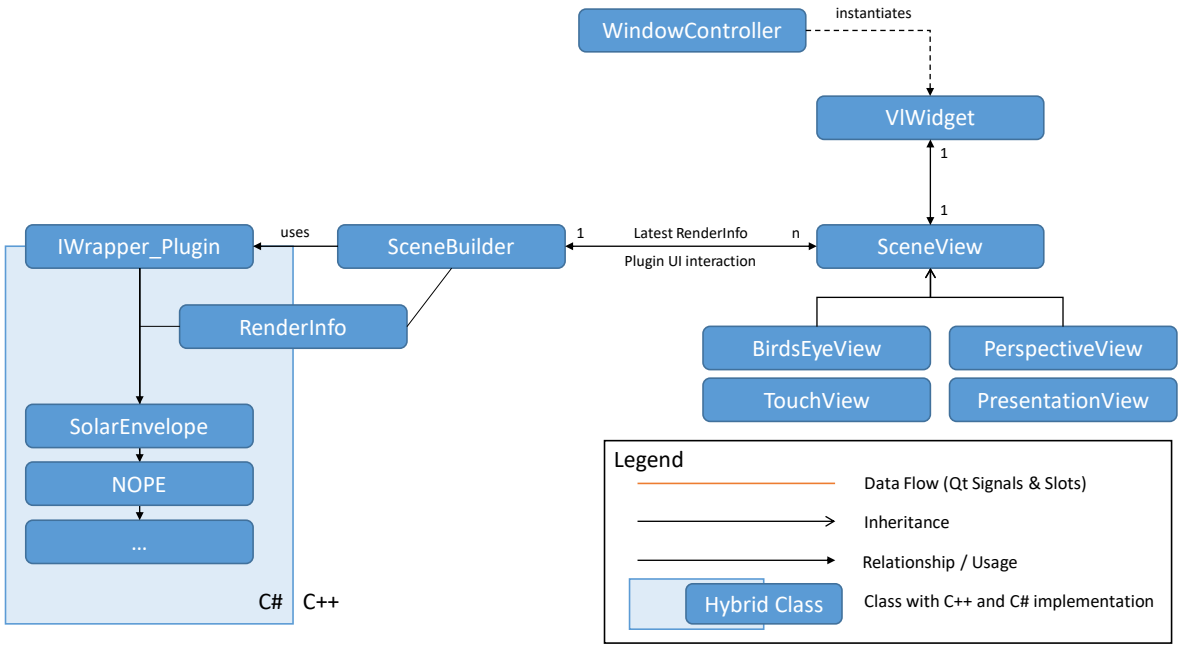

Figure 1: Simplified class overview

VlWidget

Represents a Window that contains a scene rendered with VL. Tries to abstract window management from detailed scene management that is done inside SceneView. Connects most of the data flow of SceneView, e.g. with Document(Controller), Gestures, etc.

SceneView

Abstract base class for all scenes rendered using VL. Manages all scene contents such as buildings, greenspaces, etc. Also handles rendering of latest plugin data obtained from SceneBuilder.

BirdsEyeView

Base class for an orthographic top-down view of a scene. Manages 2D camera setup. Separated from TouchView to allow usage of the rendering pipeline without core CDP-specific overlays (powerwall project).

PerspectiveView

Base class for a perspective view of a scene. Manages 3D camera setup. Separated from PresentationView for the same reasons as above.

TouchView

Inherits BirdsEyeView, implements rendering of Table Screen overlays such as compass and menu layer.

PresentationView

Inherits Perspective View, implements Presentation Sketching functionality migrated from the PresentationFramework project.

SceneBuilder

Main interface for usage of plugins. Replaces the old PluginController. Launches and updates plugins, distributes latest plugin data to all current SceneViews.

RenderInfo

Main container class for plugin data. C# side used by plugins, converted to C++ in WrapperPlugin (PluginController.cpp for some reason), and subsequently used by SceneBuilder and SceneView for rendering.

Plugin Interface

As teased above, plugins are now maintained solely by the SceneBuilder, which has only a single instance active at a time. SceneBuilder takes care of loading plugins, starting them, updating them with new information such as finger taps or physical buildings and relaying UI interactions from C++. It also regularly calls their rendering routines to update to the newest plugin state. This state is then cached and distributed to all connected SceneViews (usually exactly two) which then update their plugin rendering and plugin UI.

Plugin UI

Since all rendering should happen on C++ side (and this is enforced by not sending the OpenGL context to plugins anymore), we had to come up with a way to abstract away a plugin's UI components from their actual UI representation. This has been done by sending UI specifications for a limited set of controls from plugin side to the SceneBuilder. The respective TouchView will then process that information into a Widget and relay any interaction back to the plugin.

Due to limitations with the Qt widget system (VL only works inside a QOpenGLWidget, and these can't have sibling widgets), we had to implement the plugin UI as separate windows, that overlay the main window. Thus we needed to add another top-level window, the PluginUiWindow, which manages the VLWidget and its plugin UI components.

PluginUiWindow

Contains one VLWidget and a set of PluginGroupWidgets associated with the SceneView of the VLWidget.

PluginGroupWidget

Represents one group of UI components for one plugin.

PluginUi

Represents all groups of UI components for one plugin. And manages their interactions for a SceneView

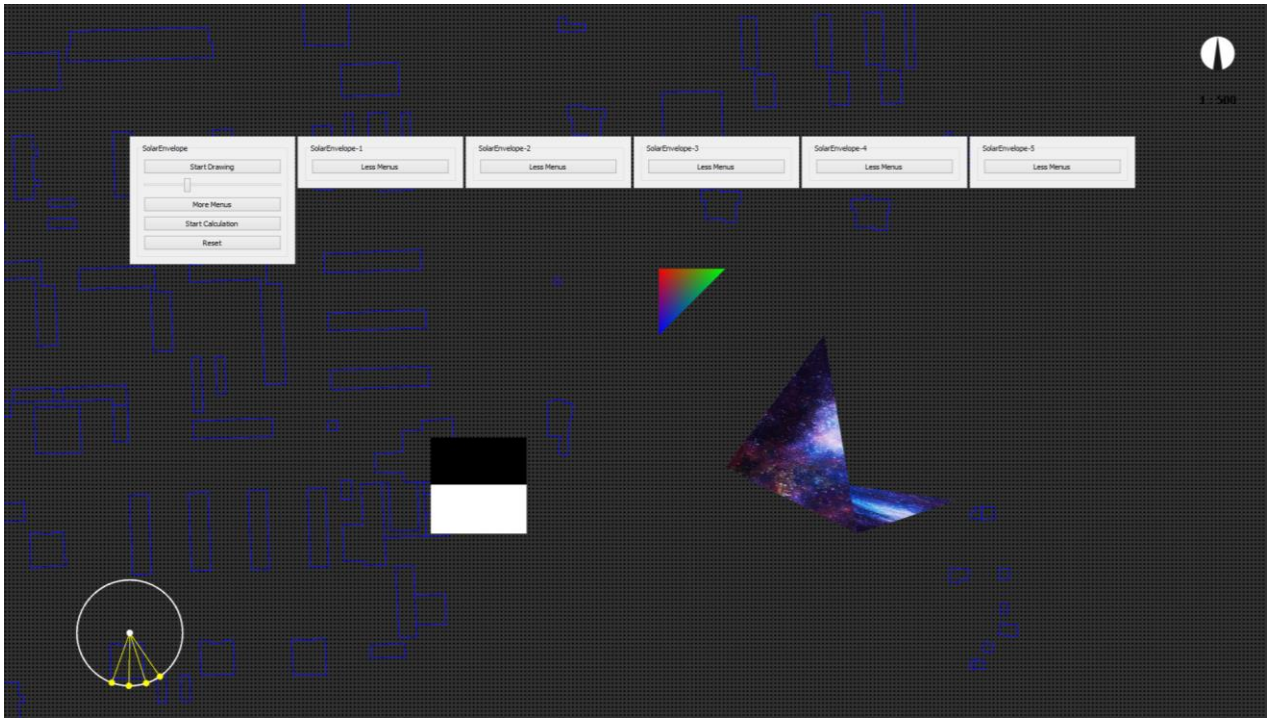

Figure 2: Sample PluginUI

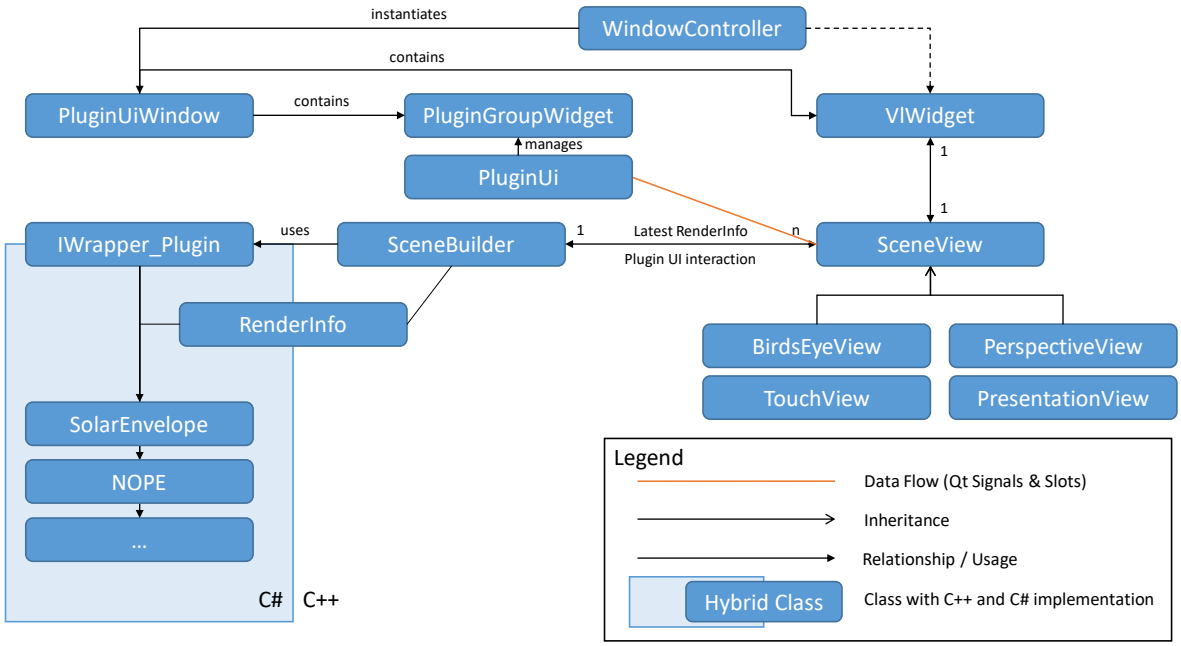

Figure 3: Class overview including Plugin UI components

Coordinate Systems

CDP uses a plethora of different coordinate systems. During the project we tried to unify those systems as much as possible. At the least we obtained an overview of when which coordinates are expected, and want to document that here.

1. Latitude and longitude: These are the most unambiguous coordinates used. However they are used only during the selection process and downloading of data from OpenStreetMaps
2. CDP coordinates are the coordinates saved inside the json file and used in the CDP::Building classes. They are calculated from latitude and longitude and range from $[0, \frac{\text{tableWidth}}{\text{mapScale}}]$ horizontally and $[0, \text{tableHeight}/\text{mapScale}]$ vertically. TableWidth is a constant of 1.5, and mapScale describes the scaling factor with respect to the actual table, e.g. 0.002 for a scale of 1:500. Thus these coordinates describe the physical offset of the table's bottom-left corner in centimeters.
3. PresentationFramework coordinates are normalized to $[-\frac{\text{tableWidth}}{2}, \frac{\text{tableWidth}}{2}]$, irrespective of the current mapScale. Additionally the tableWidth used here is different from the original tableWidth in that it is statically scaled up by a factor of 100. These coordinates range from $(-75, -45)$ to $(75, 45)$. This transformation happens in SceneView::refreshTableTransform()
4. Plugins expect building coordinates in a range of $(0, 0)$ to $(1920, 1080)$, i.e. as pixel locations on the Table Screen. To minimize the effort required to migrate plugins to the new pipeline we kept that invariant in place and transform to PresentationFramework coordinates inside Wrapper_Plugin::convertPluginContext() (PluginController.cpp)

SceneView

SceneView is the central class for scene management. Most methods should be self-explanatory.

Synchronizing State

Usually there will be multiple SceneView-derivatives active at the same time – e.g. for CDP a TouchView and a PresentationView. It is usually desired that both display the same state, i.e. if a building is hidden by a table gesture, it should also disappear from the Presentation Screen. To do this, a visitor pattern is employed. For each other derivative SceneView::visit() will be called once at scene setup. The suggested way to synchronize state is to add a signal and slot for each action that should be mimicked – and connect respectively where necessary. Currently hidden buildings are the only thing synchronized. This can easily be extended to streets or other things.

From Plugin To Screen

Overview

While the previous part of our documentation specified how the unified rendering was achieved, this part describes how the rendering data is sent from the plugins to the final scene and explains how to write a plugin that is conform to our novel rendering pipeline.

To guarantee a unified scene rendering, all plugin rendering data must be sent to the main framework. The following illustration describes all the steps and data flow from the plugin to the final rendering call in the ***SceneView***.

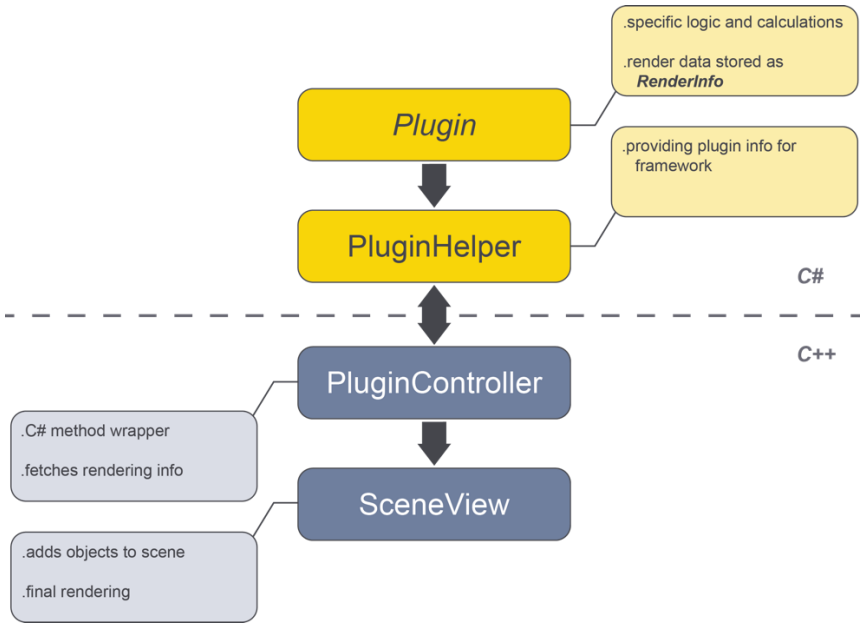

Figure 4: Overview of the data flow through the rendering pipeline

Each ***plugin*** has at least one main class on the C# side of the CDP, containing the individual logic and calculations.

If certain results need to be visualized or rendered, all the corresponding information is stored as the type ***RenderInfo***, a class that summarizes all the possible rendering data types.

The ***PluginHelper*** provides –as the name suggests- helper methods, that can be called from the C++ part of the project. Which brings us to the ***PluginController***, a C# wrapper that calls the methods provided by the *PluginHelper* and fetches the data produced by the plugins, like for example the rendering information.

This information is now constantly accessed and processed by the ***SceneView*****,** where the shader and effects are set and the final rendering is done.

Rendering Info

The class ***RenderInfo*** contains all the information that is sent from the plugin to the final rendering method. This rendering info includes *primitives*, *meshes*, *GUI elements* and some *boolean values* or rendering settings that can be defined by the plugin (e.g. if the streets or buildings should be visible in the plugin). An illustration of those is shown in **Figure 5**.

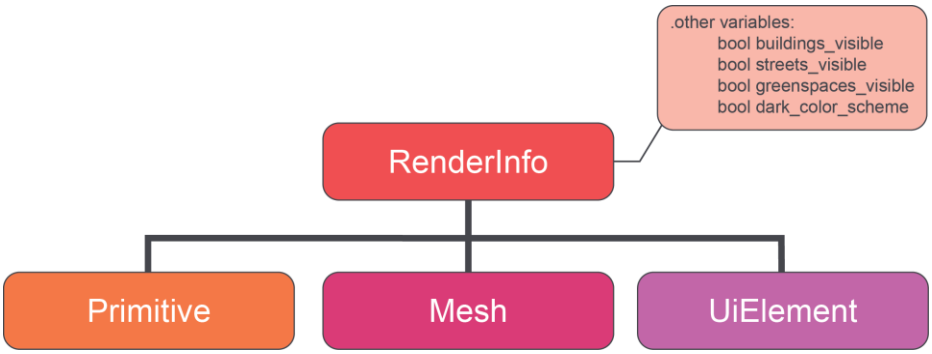

Figure 5: Overview of the data that is passed through the rendering pipeline

In general, the structure of the main rendering information types corresponds to the one defined by OpenGL. Especially when it comes to the attributes or how the rendering data is handled later. Either way, there are some differences and a further description is needed. This should be provided in the following.

Primitives

Our primitives directly correspond to the definition of the OpenGL primitives.
Primitives describe how a stream or list of defined vertices should be handled.
Each primitive is therefore specified by its type, an attribute, the position of the vertices and their color.

Figure 6 gives an overview over the different types that are implemented in our rendering pipeline. The representation and definition of color and vertices are the same and their representation and handling is automatically different for each type.

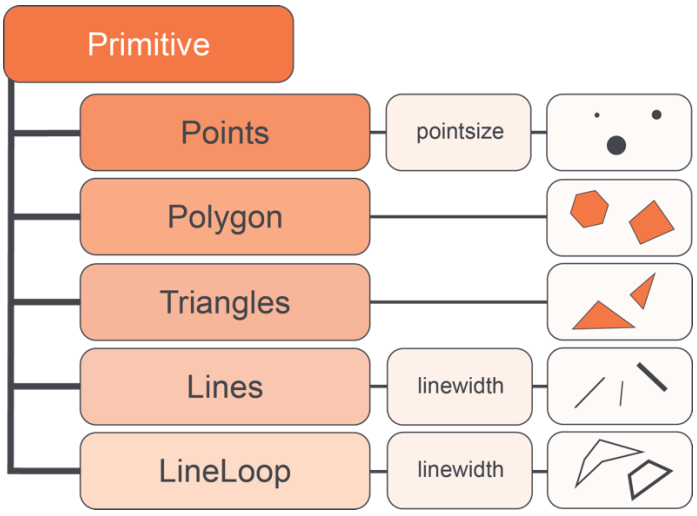

Figure 6: All implemented primitives with their respective attribute (if existent) and look

If the primitive type is ***points***, then each vertex in the stream is defined as one single point, with a defined point radius and color.
A ***polygon*** connects all vertices into one element and fills the inner area with interpolated defined colors.

Similarly work ***triangles***, but they take only every three vertices inside the stream at once and create a triangle.
Lines connect every two vertices to a simple line, which is just defined by their colors and a line width, while a ***line loop*** takes arbitrarily many vertices and connects all of them.

If another primitive –that is not yet defined- is needed during plugin development, it can easily be added inside the *primitive* class. This will be explained in more detail in the section "*Plugin Creation – How To?".*

Meshes

Another possibility of representing vertices are ***meshes***, where many coherent triangles are forming a *triangle mesh*.
In our implementation, a mesh can have a color or a texture. A color is assigned to each vertex and is then interpolated on the face, while the UV-coordinates of a texture are assigned to each vertex and define how the texture is spanned on the mesh.

Textures themselves can be loaded as an image (of various types) from a given path or are passed as a defined byte array. The handling of the textures is corresponded to the one in OpenGL or the Visualization Library.

Another variable that is passed on in the rendering pipeline is the priority offset, which sets the rendering priority of an object.

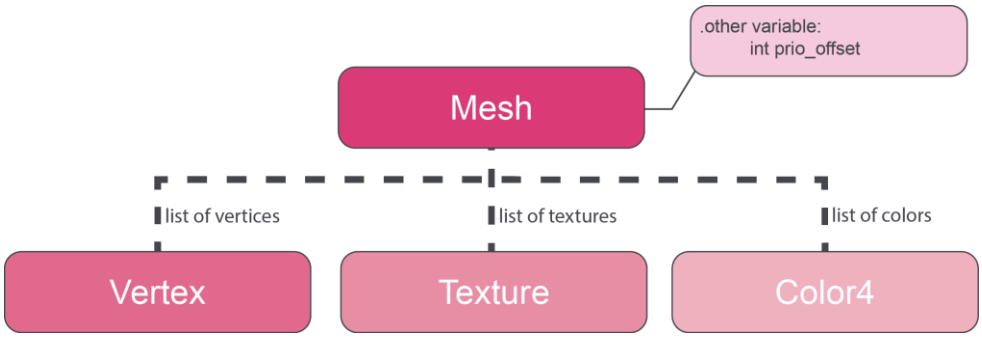

Figure 7: Overview of all mesh attributes

User Interface

The final data that needs to be passed through the rendering pipeline is the one of the plugin's user interface. For that reason, the elementary UI controls ***button***, ***slider***, ***checkbox*** and ***radio button*** are implemented with their basic functionality and can be found in the class ***UiElement***.

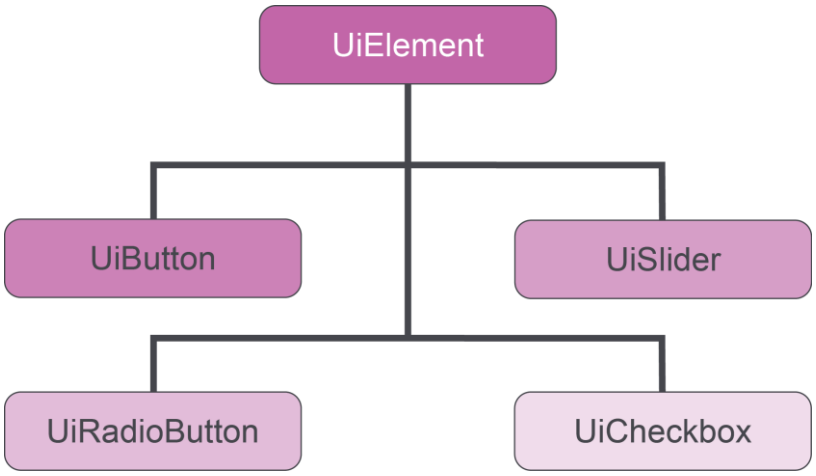

Figure 8: Overview of all implemented UI elements, whose parent in UiElement

Others

The other data sent to the final rendering method is the information about the visibility of non-plugin-specific objects, like ***buildings***, ***streets*** and ***greenspaces,*** as well as the ***color scheme*** (dark or bright) of the visible scene. An overview of how these Boolean values affect the look of a plugin can be found in the following images.

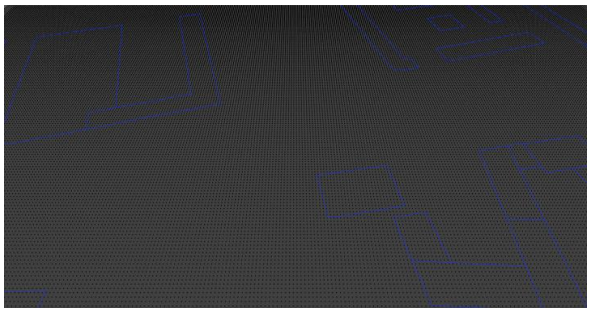

Figure 9: Plugin scene without any other objects

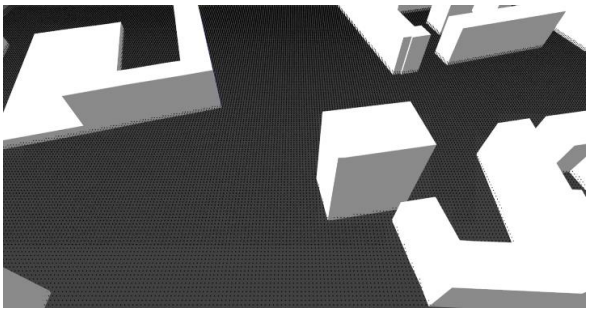

Figure 10: Plugin scene with visible buildings

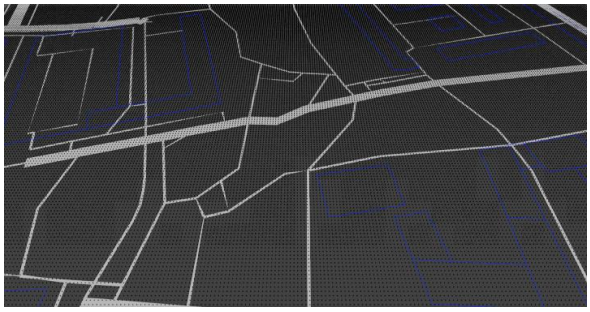

Figure 11: Plugin scene with visible streets

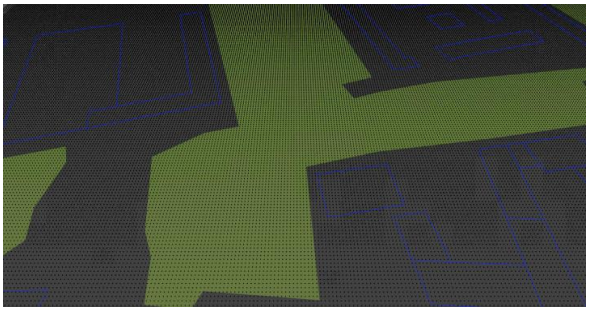

Figure 12: Plugin scene with visible greenspaces

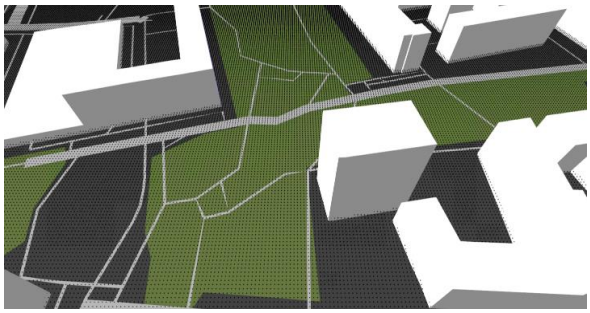

Figure 13: Plugin scene with all visibility values set true

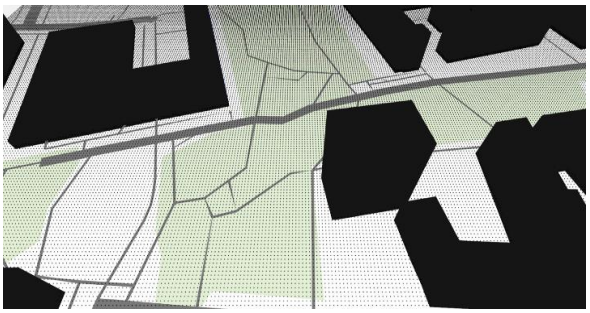

Figure 14: Dark color scheme is disabled

Plugin Creation – How to?

In this part, we want to talk about what needs to be done to create a plugin or rewrite an old one and make it work with the new rendering pipeline.

Coordinate System

The coordinate system of a plugin is derived from older implementations, where the plugins were solely developed for the table, not the screen. This means, that the x- and y-axis correspond to the width and height of the table and the z-axis describes the depth, which goes upward the table.

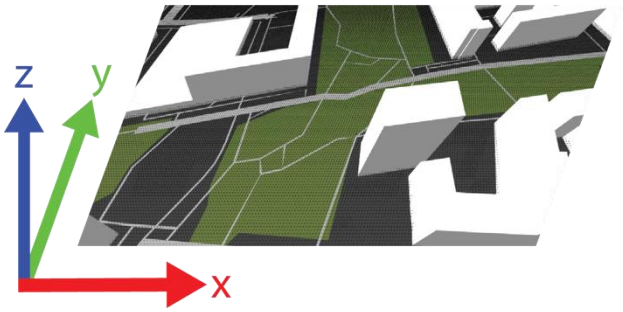

Figure 15: A plugin's coordinate system

The y- and z-values are however flipped later when the plugin is rendered in the C++ part, so the values fit the default coordinate system.

As was mentioned before, plugins expect and return their data coordinates in a range of (0,0) to (1920, 1080). The hard-coded transformation, as well as the flipping is handled in the method `convertVertices()` inside the `SceneView.cpp`.

These values can and should be adapted when more plugins are migrated or new ones are created that do not want to work with that fixed range.

Rendering Data

In general, the basic concept of handling the rendering data is the same as in OpenGL. So, if something was not mentioned here, then it is probably not implemented yet or a better explanation can be found in the OpenGL or VL documentation.

List of Primitives

A ***primitive*** always expects a list of *vertices* and *colors*, a *priority offset* and –if needed- an *attribute*. If no colors are defined, they are rendered in a default color, which is black. It is redundant to define more colors than vertices. If that happens, the console prints out a warning as this is no error, but unnecessary. If there are less colors defined than vertices, the rendering pipeline handles it the same way as OpenGL and assigns the last defined color to the remaining vertices. A more detailed description of all primitives can be found in `Primitives.cs`.

After creating and defining the individual primitives in your plugin, they all need to be stored in a single list `List<Primitive>` before they can be added to the ***RenderInfo***.
If another than the defined primitives is needed, it can simply be added inside the `Primitive.cs` class. The code should be self-explanatory.

List of Meshes

A ***mesh*** needs a list of *vertices* and a list of *colors* or *textures*. Even though you can add multiple textures to one object, currently only one is rendered. If needed, this functionality can be extended.

Like the primitives, all individually defined meshes finally need to be added to a single list `List<Mesh>`.

One of the differences to primitives is the representation of the ***vertices*** of a mesh.
Beside a position, a mesh vertex also has ***UV-coordinates*** for further texture definition.

More about the implementation can be found in the `Mesh.cs` class itself.

Textures

Another big topic in the context of meshes are ***textures***. Their implementation can be found in the class `Texture.cs`.

A texture needs the following ***attributes***:

- texture dimension/target
- mipmap values (magFilter, minFilter)
- texture format
- width, height
- texture/image type
- 3D byte array OR image path

For more information about the general meaning of those attributes, you should refer to the official OpenGL or VL websites as they give a more detailed overview.

For the values that are represented as ***enumerations***, the integer values correspond to the ones defined by the Visualization Library, not OpenGL.

When loading a texture from a path, it is important to store this image in the assigned folder `../resources/presentation/textures`.

Note again, that only one texture can be handled per mesh, even though a list of textures can be defined. This was done to make the adaption for further additions easier.

UI Elements

Other objects that need to be rendered are ***UI elements***. The already introduced elements, *UiButton*, *UiSlider*, *UiRadioButton* and *UiCheckbox* and their implementation is found in the class `UIElement.cs`. If UI elements should be added to a plugin, they are (like meshes or primitives) stored in a List and added to the *RenderInfo.*

Other Values

The visibility values are by default all set *false* and need to be set individually. The color scheme is dark by default.

Rewriting a Legacy Plugin

If an old plugin should be refactored to work with the new rendering pipeline, it is important to remember that the logic and calculations are not changed. It's just the data representation inside OpenGL statements that are not rendered immediately and are replaced by lists in which all the information is stored.
The order of how those elements are stored inside the list can and should remain the same as before, as the data is processed the same way.

Render Info Representation in C++

To get the rendering information into the C++-written framework, the `PluginController.cpp` accesses the relevant plugin information through the method `Wrapper_Plugin::GetPluginRenderInfo()`.

The representation of the rendering information in C++ is similar to the one in C#, which means that it is mainly stored as vectors without any special functionality, just to pass on the data.

Rendering in SceneView

The main plugin rendering action is happening in the `SceneView.cpp` and makes use of different VL classes, like the `Rendering` class. The rendering objects are stored as `Actors` inside an `ActorTree` and added to the general `SceneManager`. The most important method for the plugin rendering is `SceneView::pluginDataChanged()`, where effects and shaders are set for each object and the data is added to the `SceneManager`.

Future Possibilities

One thing that should be done in the future, is **migrate more plugins** to showcase the full potential of the pipeline and possibly add more functionality (for example, develop ways of how multiple plugins should be displayed).

Also, having multiple ways of displaying a plugin (2D and 3D) gives a new possibility of **extending a plugin's capabilities**.

Furthermore, the general architecture of the CDP framework, as it requires for data to be often sent from C++ to C# and vice versa, which can cause **redundant data flow**. This **should be reduced**.

Another issue with the CDP framework is, that there is a lot of unused or unnecessary code, so a general **clean-up of the code** is required.

Finally, it should be switched to **another 3D rendering pipeline**, as the current used VL version is outdated and generally barely used and without strong community or development.

Links

CDP	//	http://cdp.ai.ar.tum.de/
Visualization Library	//	http://www.visualizationlibrary.org/docs/1.0/
OpenGL	//	https://www.opengl.org/