View Configuration

Once the project loads, the application must determine how to configure the client view. A single project may contain multiple datasets. Configuring the view in the client involves selecting which datasets should be used and how will they be combined and presented in the client.

There are two main ways to set up the client view:

1. Use a predefined View Entity in the project.
2. Create a new view definition on the client by instantiating view metadata.

It is possible to mix these two approaches, that is, load the metadata for a predefined view, further modify it on the client, and the using the modified metadata for the view.

Using predefined views

The project may already contain a file entity of type “VIEW” that holds the desired configuration. The client allows to load any predefined view by calling:

```
MyView.SetViewId(“539CE7EFAA1F4F0BA6CF037F936BF035”);
```

Where the parameter is the ID for the view in the project’s entity model.

If this ID is unknown, it is possible to get a list of all available pre-defined views in a project. To do this the application may call the “GetEntities” function to the client view object:

```
var views = MyView.GetEntities(VoxelFarm.VoxelFarmEntityType.View);
```

The parameter is the type of entity to return. Only entities of that type will be returned by the function.

This function returns a list of dictionaries, where each dictionary represents a different entity in the project.

The application can use the “ID” key to obtain the unique identifier for the entity, in this case the view. This is the identifier that would be provided to the “SetViewId” function.

Creating view metadata

The C# Client uses view metadata objects to describe how different datasets in a project can be combined.

The view metadata is always an instance of the “VoxelFarmViewMeta” class.

An instance of VoxelViewMeta will contain a list of inner components. Each component represents a different type of spatial data that should appear in the view. For instance, view that shows a terrain surface with ortho-imagery laid on top of it, would contain two different components: the terrain surface geometry and the ortho imagery set.

All components inherit from the “VoxelFarmMetaComponent” class. The available components in this version are:

1. VoxelFarmMetaPointCloud. Used for point clouds.
2. VoxelFarmMetaSurface. Used for volumetric entities (terrain, block models, etc.) and indexed meshes.
3. VoxelFarmMetaOrthoImagery. Used to apply imagery to surfaces.

The VoxelFarmMetaSurface class not only represents volumetric components, but also components that are the result of volumetric operations between components. To combine different surfaces into a single operation, use the “VoxelFarmMetaSurfaceOperation” class.

For volumetric surfaces that are not the result of surface operations, use the “VoxelFarmMetaLayerStack” class. An instance of this class represents a surface that is the result of multiple volumetric layers added together. The layers can be of any of the following types:

1. VoxelFarmMetaLayerHM. Represents voxel terrain.
2. VoxelFarmMetaLayerBM. Represents a voxelized block model.
3. VoxelFarmMetaLayerPython. Represents a Python voxel generator.

The following example creates view metadata for a terrain with ortho imagery applied to it and loads it into a client view object:

```
var components = new List<VoxelFarmMetaComponent>();
var componentMasks = new List<ushort>();
var images = new VoxelFarmMetaOrthoImagery {Id = item["ID"], Diffuse = hasOrtho, Normal = hasNormals, Config = item["runtime"] };
components.Add(images);
componentMasks.Add(0x01);
var hmLayer = new VoxelFarmMetaLayerHM { Id = item["ID"], Config = item["runtime"] };
List<VoxelFarmMetaLayer> layerStack = new List<VoxelFarmMetaLayer>();
layerStack.Add(hmLayer);
var layers = new VoxelFarmMetaLayerStack(layerStack);
var surface = new VoxelFarmMetaSurfaceOperation { OpId = VoxelFarmMetaID.META_OP_NONE, OpA = layers, OpB = null, Images = images };
components.Add(surface);
componentMasks.Add(0x02);
var viewMeta = new VoxelFarmViewMeta { Components = components, ComponentMasks = componentMasks };
MyView.SetViewMetaData(viewMeta);
```

Getting metadata for a predefined view

If required to modify a predefined view, without saving any changes to it on the server, the application may obtain a local copy of the view definition by calling the “GetViewMetaData” function:

```
var viewMetaData = MyView.GetViewMetaData();
```

The view metadata can then be set back into the client view by calling the “SetViewMetaData” function:

```
MyView.SetViewMetaData(viewMetaData);
```

Revision #1
Created 17 March 2025 20:17:05 by admin
Updated 17 March 2025 20:19:49 by admin