C# Client Library

Introduction

The Voxel Farm solution includes a thin C# client library. This client allows applications running on .NET to connect and access data hosted by Voxel Farm servers.

The C# Client Library allows the application to create a “view” of content in the Voxel Farm platform. The library manages how the data in this view is streamed from the server to the client. This can be used to create a simple application that just requests data for exporting or further analysis, to complex rendering applications that must make efficient use of network and local resources as users move quickly in space, change queries and viewing parameters, etc.

The C# Client is the foundation of the Unity3D client. For a complete example of how to build a rendering application using this client, please refer to the included Unity3D example. This section also contains examples of how a simple data retrieval application can be built with this library.

In order to use the C# Client Library, it is necessary to add the VoxelFarmCloudLibrary.dll to the C# application as a reference.

Threading Model

The C# Client is designed to operate in high-frequency, real-time systems like VR rendering. It can only be used asynchronously.

The C# client uses worker threads to perform operations like downloading and decompressing data in the background. Before anything else, the application must initialize this pool of worker threads by calling:

```
VoxelFarm.VoxelFarmWorkPool.Start(NumberOfThreads);
```

Where “NumberOfThreads” is an integer specifying how many background threads should be used.

The “VoxelFarmWorkPool” object can be used to start new tasks either in background threads or in the main thread by calling the “RunInBackground()” or “RunInMainThread()” respectively. These functions take an action as a parameter, which will be queued and will be executed as soon as possible. Since the functions only insert the action into a queue, they return immediately.

The application is responsible for triggering the processing of pending actions in the main thread. To achieve this, the application must call the “RunNextMainThreadJob()” method in the “VoxelFarmWorkPool” object. Applications like client-side report programs will likely do so in loops that wait until the work is done.

It is up to the application coder to enforce thread safety for the behavior inside these actions. In general is it possible to achieve thread safety without using expensive critical sections by making background threads produce its own copy of the result data, and then passing the copy to an action in the main thread that will collate the copy with the final result. Since it is only the main thread accessing the results, all operations remain lock-free.

This code shows an example of this approach:

```
VoxelFarmWorkPool.Start(10);
string finalResult = "";
const int Iterations = 100;
int completedIterations = 0;
for (int i = 0; i<Iterations; i++)
{
   VoxelFarmWorkPool.RunInBackground(() =>
```

```
    {
        string partialResult = Guid.NewGuid().ToString();
        VoxelFarmWorkPool.RunInMainThread(() =>
        {
            finalResult += partialResult;
            completedIterations++;
        });
    });
}
while (completedIterations<Iterations)
{
    VoxelFarmWorkPool.RunNextMainThreadJob();
}
```

This example will initialize the work pool with 10 threads. Next, it will compute 100 random strings (using GUIDs) in parallel. Whenever a thread has computed the new string, a new task is scheduled to run in the main thread so the string can be added to a single final string which will eventually contain the 100 random strings

Coordinate Systems

When working with Voxel Farm projects, it is possible to encounter up to three different coordinate systems. These systems are:

1. Project’s native coordinate system. These are the real-world coordinates, typically involving an Earth ellipsoid model (WSG84, etc.) a projection system (UTM, Mercator, etc.) and other additional parameters. In most cases, these are right-hand systems with Z coordinate going up.
2. Voxel Farm’s internal coordinate system. This is a coordinate system that is used by all Voxel Farm components. It is equivalent to OpenGL’s system (right-handed, Y-up) but it does not contain negative coordinates.
3. Rendering engines impose their own coordinate system, for instance, Unity is left-handed, Y-up.

Whenever necessary, it is possible to convert from project coordinates to Voxel Farm coordinates, and vice versa, by using the VoxelFarmAffineTransform class.

The class must be initialized with the project’s coordinate system and the voxel size. Once initialized, the “VF_TO_WC()” and “WC_TO_VF()” functions provide coordinate conversions from one system to another.

Spatial Indexing

A grid is assumed to divide the world into equally-sized 3D cells. Each cell measures VoxelFarmConstant.CELL_SIZE units along each direction.

From a given 3D point in Voxel Farm coordinates, it is possible to determine the coordinates of the cell containing the point. This is simply done by dividing each coordinate by VoxelFarmConstant.CELL_SIZE and keeping the integer part. Each cell can be uniquely described by its integer coordinates in this grid. This grid is known as VoxelFarmConstant.LOD_0 because these are the smallest cells possible.

The engine then assumes the existence of an array of additional grids, each one having cells with dimensions twice the size of the previous grid. These are known as the LOD 1..N cells.

Since the grids are aligned, it can be assumed that one cell at LOD(N) will contain eight cells at LOD(N-1).

Any cell, no matter on which LOD grid, can be uniquely described by four coordinates:

1. LOD of the cell
2. X index of the cell in the LOD grid
3. Y index of the cell in the LOD grid
4. Z index of the cell in the LOD grid

To uniquely represent the cell in the index system, these four coordinates are packed into a single 64bit identifier. These identifiers are typically referred to in the API as “cell Ids”.

Voxel-based objects will use an additional grid, which divides each cell into voxels. The number of voxels in a cell is defined in VoxelFarmConstant.BLOCK_DIMENSION.

Creating a View

In order to access spatial information from the Voxel Farm, the application will have to create at least one instance of the “VoxelFarmView” class.

```
MyView = new VoxelFarm.VoxelFarmView();
```

The view object will connect to the server to request data based on user input and camera location. The application must provide the server information by calling the “SetServer” method:

```
MyView.SetServer(“http://localhost:3983”, “localhost”, 3333, false);
```

The first parameter is the URL for the server. If no port is included in the URL, port 80 will be assumed. HTTPS URLs are supported. The second and third parameter are the address and port for the Content Server. The last parameter indicates whether SSL should be used for the Content Server connection.

A single Voxel Farm deployment will host many different projects. The application must also define which project should be used by providing the project’s ID:

```
MyView.SetProjectId(“mytestprject001”);
```

At this point, we can ask the view to load the project’s information from the server. This is accomplished by calling the “LoadProject” method. This method takes an action function as a parameter, this action will be called when the project load operation completes:

```
MyView.LoadProject((HttpStatusCode code) =>
{
  // code contains a System.Net.HttpStatusCode for the result of the request
});
```

The “code” result parameter passed to the callback is the HTTP code returned by the server. If the server requires authentication and the default user credentials are rejected, this code will return HttpStatusCode.Unauthorized. In this case, the application can prompt the user for new credentials. To set the credentials, call the “SetCredentials” method:

```
MyView.SetCredentials(NewUserCredentials);
```

Note that it is necessary to call the “LoadProject” again, so the new credentials are used.

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);
```

Using Views for Rendering

The C# Client Library allows to integrate the Voxel Farm platform into existing rendering environments.

The VoxelFarmView object in the library has the concept of a “focus” scope. Typically, a 3D application will have a camera, and the camera is expected to move often unpredictably and expose different regions of the 3D content. The “focus” point for a Voxel Farm View is defined as a region of space the application wants to see in higher detail. The VoxelFarmView object will load higher resolution data in the focus region, and it will lower the resolution of the data as it becomes more distant from the focus scope.

The VoxelFarmView also natively exploits the spatial and temporal coherence of the spatial models. The data used for a particular focus setting could be mostly reusable from the next focus point. The library makes sure only the portions that changed or were not in focus before will be requested from the server and loaded. The library also keeps an internal ephemeral cache that will remember some regions of space for a while, since very often users return to a previous vintage point.

To best exploit these advantages, the application must provide a special interface to the view. This interface will be called by the view to notify new content is ready for use. This is the “IVoxelFarmRenderer” interface. The application must supply its own implementation of the interface by calling the “SetRenderer” method:

```
MyView.SetRenderer(myRenderer);
```

Implementing IVoxelFarmRenderer

The “IVoxelFarmRenderer” is used by the application to receive notifications from the C# Client. This section describes the methods of this interface. For an example of a working IVoxelFarmRenderer implementation, look at the VoxelFarmUnityView class in the Unity example.

```
void SetVoxelFarmOrigin(double x, double y, double z);
```

The client notifies the application the origin of the project in Voxel Farm coordinates.

```
IVoxelFarmCellResource BakeCellMesh(ushort clusterId, ulong cellId, byte layer, byte submesh, uint[] faces, float[] vertices, float[] normals, byte[] colors);
```

The client requests the application to create a mesh resource for a given section of the scene. The clusterId, cellId, layer and submesh arguments identify the mesh section. The faces, vertices, normal and colors arrays contain the actual mesh information.

```
IVoxelFarmCellResource BakeCellTexture(ushort clusterId, ulong cellId, byte layer, byte textureType, ushort textureSize, byte[] textureData, byte textureFormat);
```

The client requests the application to create a texture resource for a given section of the scene. The clusterId, cellId, layer and textureType arguments identify the texture section. The textureSize argument specifies the dimensions of the texture in pixels. The textureData array contains the actual RGB information. The textureFormat argument specifies in which format the texture is stored.

```
IVoxelFarmCellResource BakeCellPoints(ushort clusterId, ulong cellId, byte layer, float[] vertices, byte[] colors, double[,] aabb);
```

The client requests the application to create a points resource for a given section of the scene. The clusterId, cellId and layer identify the point cloud section. The vertices and colors array contain the actual point information. The aabb argument represents a bounding box for the points section.

```
void NotifyCellDestroyed(ulong cellId);
```

The client notifies the application a given scene cell is no longer used and can be disposed from memory.

```
void NewScene(Dictionary<ulong, bool[]> nextSceneData);
```

The application is notified a new scene configuration has started.

```
void SwapScene(Dictionary<ulong, bool[]> nextSceneData, VoxelFarmCellCache cellCache);
```

The application is notified the new scene is complete and has replaced the previous scene configuration.

```
void Clear();
```

The client asks the application to clear all resources.

```
void Begin();
```

The application is notified new viewing conditions have started (usually as a result of setting new View Metadata)

Setting view focus

Once the view is configured for rendering, the client may start calling the “SetFocus” function. This would happen every time the user moves the viewing position. The “SetFocus()” function instructs the view object to produce a new representation of the data, where the provided coordinates

appear in an adequate level of detail:

```
MyView.SetFocus(x, y, z, 0.0, 0, 3);
```

The first three parameters are the (x, y, z) coordinates of the focal point for the view. The fourth argument is the distance from the observer to this point, in this case the “0.0” makes the system assume the observer is right at the focal position.

All spatial units in this call are in Voxel Farm coordinates, to set the focus region using project coordinates, the caller must transform the coordinates first into Voxel Farm space by using the affine transform object that is already initialized in the view object. This transform is held in the “AffineTransform” property.

The fifth argument can be either zero or one, where a value of one increases the view resolution by a factor of 2. The last argument is the radius of the first level of detail.

Per-frame work

At each frame the, the application may call the “Update()” function to the view:

```
MyView.Update();
```

By calling Update, the application asks the view to discover if a new scene configuration is necessary and whether new portions of the data need to be requested.

In most application frameworks, like Unity3D, Unreal Engine 4 and OpenGL, certain operation involving hardware resources must all execute in the same thread. This is usually the main application thread or a rendering thread.

For this reason, an application using the C# client will also be responsible for making calls to “VoxelFarmPool.RunNextMainThreadJob()” from a thread the application considers to be the one in charge of managing resources:

```
VoxelFarmPool.RunNextMainThreadJob();
```

In a system that performs visualization, this function would typically be called every frame. The Voxel Farm client will attempt to minimize the time spent in these calls but since callbacks to the IVoxelFarmRenderer interface will be happen in this thread, it is up to the application to provide the best performance possible.

Example of IVoxelFarmRenderer

The following code listing shows a complete console application that uses the C# Client and defines the IVoxelFarmRenderer interface for callbacks:

```
using System;
using System.Collections.Generic;
using VoxelFarm;
namespace ConsoleApp1
{
    class MyDummyResource : VoxelFarm.IVoxelFarmCellResource
    {
        private string Name;
        private ulong CellId;
        public MyDummyResource(string name, ulong cellId)
        {
            Name = name;
            CellId = cellId;
        }
        public void Release()
        {
            Console.WriteLine("Released " + Name + " for " + CellId);
        }
    }
    class MyRenderer : VoxelFarm.IVoxelFarmRenderer
    {
        public IVoxelFarmCellResource BakeCellMesh(ushort clusterId, ulong cellId, byte layer, byte submesh, uint[]
faces, float[] vertices, float[] normals, byte[] colors)
        {
            Console.WriteLine("Received mesh " + submesh + " for " + cellId);
            return new MyDummyResource("Mesh", cellId);
        }
        public IVoxelFarmCellResource BakeCellPoints(ushort clusterId, ulong cellId, byte layer, float[]
vertices, byte[] colors, double[,] aabb)
        {
            Console.WriteLine("Received points for " + cellId);
            return new MyDummyResource("Points", cellId);
```

```
    }
    public IVoxelFarmCellResource
BakeCellTexture(ushort clusterId, ulong cellId, byte layer, byte textureType, ushort textureSize, byte[]
textureData, byte textureFormat)
    {
        Console.WriteLine("Received texture for " + cellId);
        return new MyDummyResource("Texture", cellId);
    }
    public void Begin()
    {
        Console.WriteLine("New view");
        if (OnNewView != null)
            OnNewView();
    }
    public void Clear()
    {
        Console.WriteLine("Scene cleared");
    }
    public void NewScene(Dictionary<ulong, bool[]> nextSceneData)
    {
        Console.WriteLine("New Scene " + nextSceneData.Count + " cells");
    }
    public void NotifyCellDestroyed(ulong cellId)
    {
        Console.WriteLine("Cell " + cellId + " deleted");
    }
    public void SetVoxelFarmOrigin(double x, double y, double z)
    {
        Console.WriteLine("Origin set to: " + x + ", " + y + ", " + z);
        OriginX = x;
        OriginY = y;
        OriginZ = z;
    }
    public void SwapScene(Dictionary<ulong, bool[]> nextSceneData, VoxelFarmCellCache cellCache)
    {
        Console.WriteLine("Scene completed");
        if (OnSceneCompleted != null)
            OnSceneCompleted();
    }
    public Action OnNewView = null;
```

```
    public Action OnSceneCompleted = null;
    public double OriginX = 0.0;
    public double OriginY = 0.0;
    public double OriginZ = 0.0;
}
class Program
{
    static void Main(string[] args)
    {
        Random rnd = new Random();
        VoxelFarm.VoxelFarmWorkPool.Start(4);
        VoxelFarm.VoxelFarmView vf = new VoxelFarm.VoxelFarmView();
        MyRenderer myRenderer = new MyRenderer();
        myRenderer.OnNewView = () =>
        {
            // we will go to the origin
            vf.SetFocus(
                myRenderer.OriginX,
                myRenderer.OriginY,
                myRenderer.OriginZ,
                0.0, 0, 3);
        };
        myRenderer.OnSceneCompleted = () =>
        {
            // we will go to a new random location
            vf.SetFocus(
                myRenderer.OriginX + 10000.0 * rnd.NextDouble(),
                myRenderer.OriginY + 10000.0 * rnd.NextDouble(),
                myRenderer.OriginZ + 10000.0 * rnd.NextDouble(),
                0.0, 0, 3);
        };
        vf.SetRenderer(myRenderer);
        vf.SetServer("http://localhost", “localhost", 3333, false);
        vf.SetProjectId("sonomadb");
        vf.LoadProject((code) =>
        {
            if (code == System.Net.HttpStatusCode.OK)
            {
                var views = vf.GetEntities(VoxelFarm.VoxelFarmEntityType.View);
                if (views.Count > 0)
```

```
                {
                    vf.SelectView(views[0]["ID"]);
                }
            }
        });
        bool terminate = false;
        while (!terminate)
        {
            VoxelFarm.VoxelFarmWorkPool.RunNextMainThreadJob();
            vf.Update();
        }
    }
  }
}
```

Using Views for Data Retrieval

The C# Client library allows a simpler interaction model for applications that just want to read spatial data from Voxel Farm servers. This is the case of a client-side report for instance, where multiple datasets can be requested and inspected by a client application, and then used to compute additional information.

Like it was the case with rendering a view, it is still necessary to set up the VoxelFarmView object with a server, a project, and the desired view configuration.

The VoxelFarmView instance provides callbacks that fire whenever information is received from the server. A client application that is interested in only receiving the data, can often do with just setting up some of these callbacks.

The data callback functions are:

OnViewStarted()	Notifies that the view has started. This happens after the client establishes a successful connection to the server.
OnReceiveVoxelData(VoxelData data)	Notifies voxel data for a cell has arrived
OnReceiveMeshData(MeshData data)	Notifies mesh data for a cell has arrived
OnReceiveTextureData(TextureData data)	Notifies texture data for a cell has arrived
OnReceivePointData(PointData data)	Notifies point data for a cell has arrived
OnCellComplete(ulong cell, VoxelFarmCellData data)	Notifies all expected components for a cell have arrived

VoxelData

This object contains voxel data for a cell. It has the following properties:

ulong cellId	Unsigned 64bit identifier for the cell.
byte layer	Identifies the metadata component that originated this data in the view metadata

float[] values	Contains the requested voxel attribute values. For each attribute that was requested in the metadata, a section of VoxelFarmConstant.BLOCK_DIMENSION^3 floats appear in this array. For instance, the values for third attribute that was requested would appear at index (2 * BLOCK_DIMENSION * BLOCK_DIMENSION * BLOCK_DIMENSION)

MeshData

This object contains voxel data for a cell. It has the following properties:

ulong cellId	Unsigned 64bit identifier for the cell.
byte layer	Identifies the metadata component that originated this data in the view metadata
byte submesh	Identifies the section of the cell that is associated to the mesh. A single cell may contain up to 7 different meshes. The first mesh is the contents of the cell, the other six (one for each side of the cell) is a seam mesh that allows the cell mesh to connect to lower LOD neighboring cells.
float[] vertices	Contains the coordinates for the vertices. These coordinates are local to the cell origin, where (0,0,0) corresponds to the origin of the cell and (1,1,1) to the farthest corner from the origin.
float[] normals	Contains the surface normal vectors for each face-vertex combination
float[] colors	Contains an RGB color triplet for each vertex
uint[] faces	Contains triangle definitions for the mesh, each entry is a reference to a vertex in the vertices array.

TextureData

This object contains texture data for a cell. It has the following properties:

ulong cellId	Unsigned 64bit identifier for the cell.
byte layer	Identifies the metadata component that originated this data in the view metadata
byte textureType	Identifies the texture type. The following types are currently supported: 0 – Diffuse texture 1 – Normal map texture

byte textureFormat	Defines the texture format: 0 – DXT1 1 – DXT5 2 – JPG 3 – PNG 4 - RGBA
ushort textureSize	Contains the dimensions in pixels of the texture. Textures for cells are square.
byte[] textureData	Contains an RGBA values (8bits per channel) for the texture

PointData

This object contains point cloud data for a cell. It has the following properties:

ulong cellId	Unsigned 64bit identifier for the cell.
byte layer	Identifies the metadata component that originated this data in the view metadata
float[] vertices	Contains the coordinates for the points. These coordinates are local to the cell origin, where (0,0,0) corresponds to the origin of the cell and (1,1,1) to the farthest corner from the origin.
float[] colors	Contains an RGB color triplet for each point

VoxelFarmCellData

This object contains all resources associated with a cell. Since a view may require multiple components to load for a single cell, this data structure will contain data results for each component in the view metadata. Individual component results can be found in the “layers” property of the object:

List<VoxelFarmCellDataLayer> Layers	Contains a list of “VoxelFarmCellDataLayer” objects, one object per component in the view metadata. Depending on the component type, the object uses a more specialized class that derives from “VoxelFarmCellDataLayer”. These are: 1. VoxelFarmCellDataLayerMesh – Contains mesh resources 2. VoxelFarmCellDataLayerOrtho – Contains texture resources 3. VoxelFarmCellDataLayerPoints – Contains point cloud resources 4. VoxelFarmCellDataLayerVoxels – Contains voxel resources

Example - Mesh Export

The following C# program connects to a Voxel Farm server and exports the terrain surface within a 3D region as a mesh:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using VoxelFarm;



namespace ExportMesh
{
    class Program
    {
        private static void Terrain2MeshOBJ(
            string webServer,
            string contentServerAddr,
            int contentServerPort,
            string projectId,
            string entityId,
            Region25D region,
            int lod,
            string outputFile)
        {
            // Create output file
            System.IO.StreamWriter fileOutput = new StreamWriter(outputFile);

            // Local variables to track OBJ output
            bool done = false;
            int totalFaces = 0;
```

```
        int vertexIndexOffset = 1;

        // Start work pool
        VoxelFarmWorkPool.Start(2 * Environment.ProcessorCount);

        // Create view
        VoxelFarmView view = new VoxelFarmView();

        // Print view debug log
        view.OnDebugLog = (message) =>
        {
           //Console.WriteLine(message);
        };

        // Also print any uncaught error
        VoxelFarmWorkPool.OnError = (Exception error) =>
        {
           Console.WriteLine(error.Message + " " + error.StackTrace);
           done = true;
        };

        // Set OnViewStarted event
        view.OnViewStarted = () =>
        {
           try
           {
              // Use the supplied region to create a set of ranges of 10x10x10 cells
              var ranges = region.breakdownRegion(lod, 10, view.AffineTransform);



              // Iterate over each range of cells
              int totalRanges = ranges.Count;
              int completedRanges = 0;
              foreach (var range in ranges)
              {
                 // Output progress message
                 int progress = (100 * completedRanges++) / totalRanges;
                 string message = (totalFaces > 0) ? String.Format("{0:0,0} triangles exported...", totalFaces)
: "Scanning for triangles...";
                 Console.WriteLine(progress + "% " + message);
```

```
            // Ask the view to load the cell range
            view.SetCellRange(range);

            // Wait until the view completes processing the range
            view.Join();

            // Check that we still have a connection to the server
            if (view.ConnectionLost() || !view.IsConnected())
            {
                Console.WriteLine("Lost connection to content server.");
                done = true;
            }
        }
        done = true;
    }
    catch (Exception error)
    {
        Console.WriteLine(error.Message + " " + error.StackTrace);
        done = true;
    }
};


// Define a callaback to handle mesh data
view.OnReceiveMeshData = (data) =>
{
    ManualResetEvent outputDone = new ManualResetEvent(false);
    VoxelFarmWorkPool.RunInMainThread(() => {

        int level, xc, yc, zc;
        VoxelFarmCell.Unpack(data.cellId, out level, out xc, out yc, out zc);
        double scale = VoxelFarmConstant.CELL_SIZE * (1 << level);
        double cellOffsetX = scale * xc;
        double cellOffsetY = scale * yc;
        double cellOffsetZ = scale * zc;

        int vertCount = data.vertices.Length / 3;
        int faceCount = data.faces.Length / 3;
```

```
        for (int i = 0; i < vertCount; i++)
        {
            var wc = view.AffineTransform.VF_TO_WC(
                new VoxelFarmAffineTransform.sAffineVector
                {
                    X = data.vertices[3 * i + 0] * scale + cellOffsetX,
                    Y = data.vertices[3 * i + 1] * scale + cellOffsetY,
                    Z = data.vertices[3 * i + 2] * scale + cellOffsetZ
                });

            fileOutput.WriteLine("v " + (-wc.X) + " " + wc.Z + " " + wc.Y);
        }

        for (int i = 0; i < faceCount; i++)
        {
            fileOutput.WriteLine("f " +
                (data.faces[3 * i + 0] + vertexIndexOffset) + " " +
                (data.faces[3 * i + 1] + vertexIndexOffset) + " " +
                (data.faces[3 * i + 2] + vertexIndexOffset));
        }

        vertexIndexOffset += vertCount;
        totalFaces += faceCount;

        outputDone.Set();
    });
    outputDone.WaitOne();
};


// Set server and project

view.SetServer(webServer, contentServerAddr, contentServerPort, false);
view.SetProjectId(projectId);

// Load the project
view.LoadProject((HttpStatusCode httpcode) =>
{
    if (httpcode == HttpStatusCode.OK)
    {
```

```
                // Find entity
                var terrainEntity = view.GetEntity(entityId);
                if (terrainEntity == null || terrainEntity["file_type"] != VoxelFarmEntityType.VoxelTerrain)
                {
                    done = true;
                    return;
                }

                // Create view metadata
                var components = new List<VoxelFarmMetaComponent>();
                var componentMasks = new List<ushort>();
                List<VoxelFarmMetaLayer> layerStack = new List<VoxelFarmMetaLayer>();
                layerStack.Add(new VoxelFarmMetaLayerHM { Id = entityId, Config = terrainEntity["runtime"] });
                var layers = new VoxelFarmMetaLayerStack(layerStack);
                var surface = new VoxelFarmMetaSurfaceOperation { OpId = VoxelFarmMetaID.META_OP_NONE,
OpA = layers };
                components.Add(surface);
                componentMasks.Add(0x02);
                var componentMaterials = new List<ushort>();
                componentMaterials.Add(0xFFFF);
                var metadata = new VoxelFarmViewMeta(components, componentMasks,
componentMaterials, null);

                // Set view metadata, this will start the connection to the streaming server
                view.SetViewMetaData(metadata);
            }
            else
            {
                done = true;
            }
        });

        while (!done)
        {
            VoxelFarmWorkPool.RunNextMainThreadJob();
        }

        fileOutput.Close();
        VoxelFarmWorkPool.Stop();
        Console.WriteLine("Export complete.");
```

```
        }

        static void Main(string[] args)
        {
            if (args.Length != 8)
            {
                Console.WriteLine("Usage ExportMesh.exe <REST URL> <Streaming Server> <Streaming Port>
<Project Id> <Entity Id> <Region> <LOD> <Output File>");
                return;
            }

            try
            {
                Region25D region = new Region25D();
                region.loadFromString(args[5]);
                Terrain2MeshOBJ(args[0], args[1], Convert.ToInt32(args[2]), args[3], args[4], region,
Convert.ToInt32(args[6]), args[7]);
            }
            catch (Exception error)
            {
                Console.WriteLine(error.Message + " " + error.StackTrace);
            }
        }
    }
}
```

The following command line tests this program over the Sta. Barbara Island dataset, requesting a mesh export for the full island at LOD 5 resolution:

ExportMesh.exe http://18.236.158.139 18.236.158.139 3333 blank 390CC9D2B934446DAEE5EFE9F7D64C4F 1,1.0000000000002274,300,-13253468.999999998,3937303,-13248860.999999998,3937303,-13248860.999999998,3932695,-13253468.999999998,3932695 5 f:\scrap\m1.obj

Note that this program requires the “vfcompression.dll” to be in the same folder as the .EXE, or be in one of the folders specified in the system PATH variable.

The program produces the following output to the console window:

0% Scanning for triangles...

4% 20,378 triangles exported...

8% 59,296 triangles exported...

12% 120,883 triangles exported...

16% 174,325 triangles exported...

20% 198,326 triangles exported...

24% 231,524 triangles exported...

28% 304,836 triangles exported...

32% 367,609 triangles exported...

36% 410,992 triangles exported...

40% 423,511 triangles exported...

44% 448,613 triangles exported...

48% 522,385 triangles exported...

52% 548,733 triangles exported...

56% 592,684 triangles exported...

60% 597,296 triangles exported...

64% 631,860 triangles exported...

68% 694,986 triangles exported...

72% 759,434 triangles exported...

76% 780,406 triangles exported...

80% 781,337 triangles exported...

84% 806,390 triangles exported...

88% 848,185 triangles exported...

92% 869,169 triangles exported...

96% 875,037 triangles exported...

Export complete.

The exported OBJ mesh looks like this: