

Unity Heat Map Plug-in

Game Engines E2013
IT-University of Copenhagen

Jacob Claudius Grooss, jcgr@itu.dk

December 11th, 2013

Contents

1	Introduction	1
2	Project Overview	1
2.1	Goal of the Project	2
2.2	Features	2
3	How To Use	3
3.1	Tracking game objects	3
3.2	Generating the heat map	4
4	Overview of the Plug-in	5
4.1	Editor	5
4.2	Plugins	5
4.3	Prefabs	5
4.4	Scripts	5
5	Testing	7
5.1	Games	7
5.2	Performance	9
6	Issues	10
6.1	Data from multiple files	10
6.2	Performance of heat map generation	10
6.3	Object names when tracking	10
6.4	Unity 2D tool support	10
7	Conclusion	11
7.1	Future work	11

1 Introduction

This report is the result of a project in the Game Engines E2013 (MGAE-E2013) course on the Games line at the IT-University of Copenhagen autumn 2013.

For the project, I have worked with Jakob Melnyk (jmel) to create a plug-in for Unity that allows the user to gather data for their games and create heat maps based on this data.

2 Project Overview

When it comes to video games, there is a lot of data available from when people are playing and all of it can be tracked. Data that involves the positions of objects, where entities died (and what they died to), where certain events were performed, etc. can also be very interesting for the developers. It can be used for statistics, influencing balance, finding bottlenecks and figure out what is interesting for the players.

While the data is important, displaying it in an easy-to-understand manner is at least as important. Using an FPS¹ game as an example, showing what weapons the players prefer to use overall, their accuracy with weapons, which enemies are killed with which weapon, etc. can be visualized properly with a table. It works because the data is numbers and the numbers are not necessarily related to where in the world the even takes place.

When it comes to positioning of any kind in a game, a table is not a good choice. A table is useful for saying how many times an event happened, but it is not very helpful when it comes to stating *where* the event happened. "Event e has happened at location x, y, z so and so many times" gives some information, but where in the game is x, y, z?

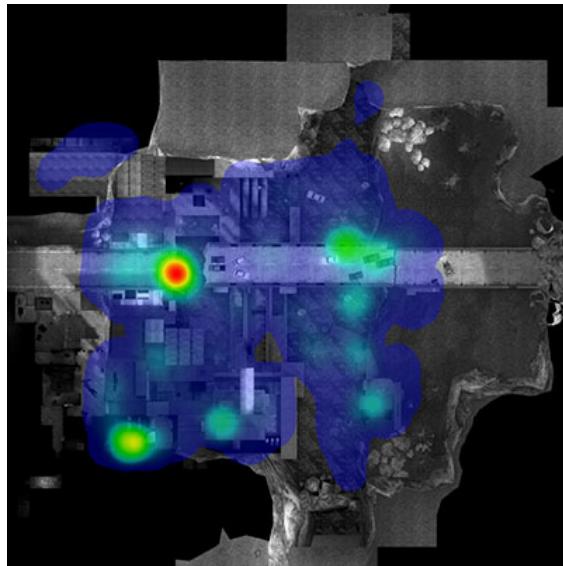


Figure 1: A heat map representing deaths in a level in Half-Life 2

This is where heat maps have their place. Using colors, they can show where - and how many times - an event has happened in the game. Heat maps are maps of the different places of the game with colored spots

¹First Person Shooter

on. The colors of the spots range from an icy blue to a dark red. The darker the color is, the more times a certain event has happened there.

Looking at **Figure 1** on page 1, one place is bright red, which means that players have died a lot at that position. This can either indicate that the place is too hard, that the players have missed something that can heighten their survival, or that the place is working as an intended difficult area. Either way, the heat map gives the developers useful information, which is easy to understand and work with.

2.1 Goal of the Project

As both Jakob and I are rather interested in game balance and mechanics, we decided we wanted to make a tool that could assist in generating heat maps. After talking it through, we decided to create the tool for Unity, for a couple of reasons.

Unity works on a set of rules. Every object in the game world has a position and every object has access to methods that are the same for all objects (such as the `Update()` method, which is called every frame on every object). This means we can assume that every object can do certain things, and base our tool on that.

Another nice thing is that plugins in Unity are actually just collections of scripts and prefabricated objects. As such, it is easy to export and import in another game. Copy the necessary scripts and objects, and you have a working heat map tool for another game.

Unity is also fairly easy to work with, when it comes to creating new tools for it. The API used in Unity is solid, and gives access to basically anything one will need. The documentation and support for Unity is also extensive, making it easy to figure out what one can do and how to do it.

2.2 Features

Having decided to write the tool for Unity, we decided on the features there should be in the tool:

- **Track anything** - The tool should be able to track any object, no matter how simple or complex the object is.
- **Generate heat map on event basis** - The tool should be able to generate a heat map based only on certain events. If all the tracked events are shown at the same time, the heat map will practically be useless.
- **Easy to use** - Finding a tool that seems to cover all your needs, just to figure out that it takes ages to set up properly is never fun.
- **Stand-alone** - Gathering and processing of data should not depend on other systems, or even connection to the internet. It should be able to work as long as the game is running.

These features were the must-have of the tool, as without them it would not be able to properly gather and visualize data.

3 How To Use

As one of our features was that the plug-in should be easy to use, we had a lot of focus on that. We have therefore boiled it down to two parts: Tracking events on objects and generating the heat map.

To use the plug-in, the package (which consists of four folders) needs to be imported. The elements the developer need for tracking and visualizing are `Scripts/HM_Tracker` and `Prefabs/HM_HeatMap`. The rest of the files are used internally to handle events, inspector UI, marker colors/transparency, etc.

3.1 Tracking game objects

To track an object, the `HM_HeatMap` script must be attached to the object. With the script attached, the developer can change the position tracker interval, save interval and which of the default events that are tracked through the object inspector UI.

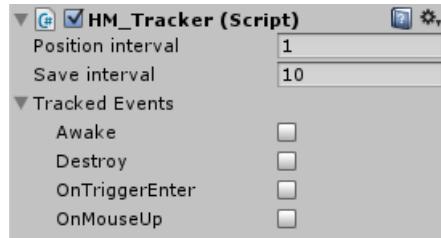


Figure 2: The tracker options.

The *Position interval* option determines the minimum amount of seconds between BreadCrumb (see below) events. It cannot be set to less than one second, as setting it lower could possibly become a memory hog and rarely result in any real gains compared to tracking every second.

The *Save interval* option is the minimum amount of seconds between the tracked events being saved to an XML file. It cannot be set to less than one second, as we felt that being able to save more often than tracking the position of an object did not make sense. It is possible, however, if the *Position interval* is set higher than one, but we chose not to limit it more than it is.

The default events the tracker supports are BreadCrumb, Awake, Destroy, OnTriggerEnter and OnMouseDown.

BreadCrumb is the name of the event that tracks the object's position, and is always activated. It is tracked in the Update method, and will be tracked at the chosen position interval.

Awake, Destroy and OnTriggerEnter are tracked every time the method is called on the object. Calling these methods is something Unity handles internally, and we can assume that without lag, these events will be tracked successfully every time they happen.

OnMouseDown happens as part of the Update method, like the BreadCrumb event. The default amount of calls to Update per second is thirty, so this event cannot be tracked more than thirty times per second.

The tracker will save all events that it is set to track, and every time the *Save interval* is reached, these events will be saved to a file named "HeatMapData/YYYY.MM.DD hh.mm.ss/HeatMapDataNameOfGameObject.xml". When the game finishes, all the individual files will be combined to a single file named "HeatMapData/HeatMapData YYYY.MM.DD HH.MM.SS.xml", which is the one that is used for generating the heat map.

Custom Events

While the tracker can track certain events by default, the developer can also use it to track custom events. There are two ways to do this.

The first way is to call the `AddEvent` method on the attached `HM_Tracker` script from another script and pass the event name and position as parameters. This is how we intend for it to work.

The second way is to modify the `HM_Tracker` script itself, to allow tracking of the custom event from inside the script.

3.2 Generating the heat map

To generate the heat map, the `HM_HeatMap` prefab should be placed in the scene. The prefab has the `HM_GenerateHeatMap` script attached, which is what generates the heat map. In the object's inspector, the developer has access to the settings of the heat map script.

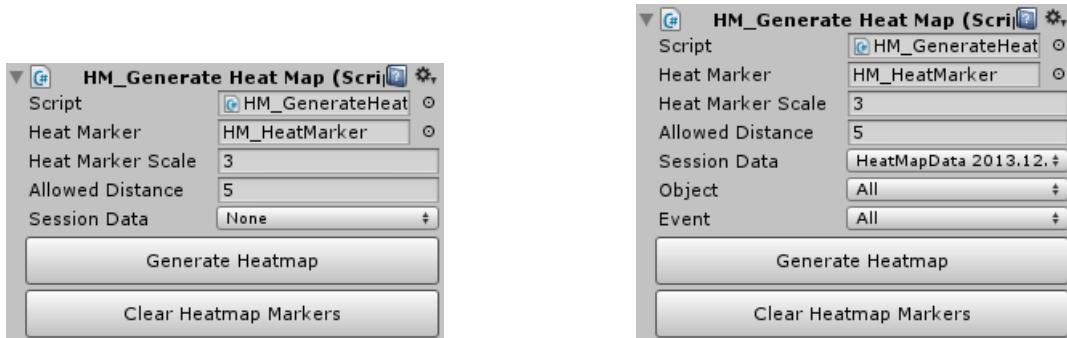


Figure 3: The options for generating heat map. Left picture is without any data selected, the right is with data selected.

Heat Marker is a field that contains the element that is used for visualizing the positions where events happened. The field should contain the `HM_HeatMarker` prefab.

Heat Marker Scale determines the scale of the objects that surrounds a tracked event (the *Heat Marker* object). *Allowed Distance* determines how far two heat markers can be from each other and still count as being "near" each other.

Session Data loads a list of the all XML files in the "HeatMapData/" folder and displays them in a dropdown menu. Upon choosing a file, the *Object* and *Event* options are loaded. *Object* contains a list of all the objects that have been found in the *Session Data* file. *Event* shows a list of all the event types that have been tracked for the chosen object.

The *Generate Heatmap* button will use the chosen parameters to generate markers for each tracked event from the chosen data. It will position the markers in the scene, and color them according to how many other markers that are nearby. Multiple combinations can be visualized at the same time, simply by pressing the button with different data selected.

The *Clear Heatmap Markers* button is used to remove the heat map again by destroying all instances of the heat marker object in the scene.

4 Overview of the Plug-in

The Scripts/HM_Tracker and Prefabs/HM_HeatMap may be the most important parts for the user, but there are a lot of other things that are used behind the scenes in order for the plug/in to work.

The plug-in contains four folders: *Editor*, *Plugins*, *Prefabs* and *Scripts*.

4.1 Editor

The *Editor* folder contains two C# files: HM_GenerateHeatMapEditor and HM_TackerEditor. These files do not extend MonoBehaviour and can therefore not be attached to any game object. Instead, they extend Editor which allows them to modify the object inspector for another script.

In order for the scripts to modify the inspector, the scripts use the GUILayout class as seen in **Code 1**. This class gives access to buttons, labels, dropdown menus, etc. It also allows the script to access different ways of grouping the created GUI elements.

```

1 GUILayout.BeginHorizontal();
2 if(GUILayout.Button("Generate Heatmap", GUILayout.Height(30)))
3 {
4     generateHeatMapScript.Generate();
5 }
6 GUILayout.EndHorizontal();

```

Code 1: The code needed to create a button in the Object Inspector and make something happen when it is clicked.

4.2 Plugins

The *Plugins* folder contains two script: Colorx and HSBCColor, both of them written by Jonatha Czeck, Graveck Interactive 2006. We use them to control the color of the heat markers in the scene, as they allow for smoother transitioning between colors.

4.3 Prefabs

The *Prefabs* folder contains two game objects: HM_HeatMap and HM_HeatMarker. The HM_HeatMap prefab was explained in section 3.2, so I will skip that for now. HM_HeatMarker is a sphere with the HM_HeatMarkerScript attached. The heat map is visualized by placing one marker at each event in the scene, and the frequency of the event in that area is visualized by changing the color of the marker.

4.4 Scripts

The *Scripts* folder contains the scripts we have written for the plug-in that do not modify the object inspector.

HM_ControlObjectScript

The HM_ControlObjectScript script is put on a dummy object that is generated when the game starts. It ensures that only one folder is created for the data gathering session, and it takes care of combining the data for the different objects when the game ends.

```

1     directory = Directory.GetCurrentDirectory() + "\\HeatMapData";
2     sessionName = string.Empty + System.DateTime.Now.ToString(@"yyyy.MM.dd
HH.mm.ss");

```

```

3         if(!Directory.Exists(directory))
4         {
5             Directory.CreateDirectory(directory);
6         }
7
8
9         if (!Directory.Exists(directory + "\\\" + sessionName))
10        {
11            Directory.CreateDirectory(directory + "\\\" + sessionName);
12        }

```

Code 2: Creating the directory for the session

HM_Event

The HM_Event script is very basic. It holds the type of event and the position at which it happened and can write this data to an XML file (see **Code 3** on page 6).

```

1 public void WriteToFile(XmlTextWriter writer)
2 {
3     writer.WriteStartElement(eventName);
4     writer.WriteAttributeString("x", position.x.ToString());
5     writer.WriteAttributeString("y", position.y.ToString());
6     writer.WriteAttributeString("z", position.z.ToString());
7     writer.WriteEndElement();
8 }

```

Code 3: The method the event uses to write its data to a file.

HM_EventTypes

The HM_EventTypes script holds the default type of events the plugin can track.

HM_GenerateHeatMap

The HM_GenerateHeatMap script takes care of generating the heat map based on the data gathered, and it creates and places the markers in the world. The data it uses is chosen through the Unity inspector options (see section 3.2).

```

1     XmlNodeList xnlNodes = xelRoot.SelectNodes("/TrackingData/" + actualObject +
2                                                 "/" + actualEvent);
3
4     // Iterate over all the event nodes and create markers for them.
5     foreach(XmlNode node in xnlNodes)
6     {
7         var x = float.Parse(node.Attributes["x"].Value);
8         var y = float.Parse(node.Attributes["y"].Value);
9         var z = float.Parse(node.Attributes["z"].Value);
10        var hm = (GameObject) Instantiate(HeatMarker, new Vector3(x, y, z),
11                                         new Quaternion(0,0,0,0));
10        hm.transform.localScale = new Vector3(HeatMarkerScale,
11                                             HeatMarkerScale, HeatMarkerScale);
11        hm.transform.parent = this.transform;
12    }

```

Code 4: Reading the tracked events and instantiating the markers.

HM_HeatMarkerScript

The `HM_HeatMarkerScript` script is placed on the `HM_HeatMarker` objects, and is used to change the color on the marker. To do this, it has to create a new material for itself and set both the color and transparency of this material (see **Code 5** on page 7).

```

1 public void SetColor(Color color)
2 {
3     var tempMat = new Material(Shader.Find ("Transparent/Diffuse"));
4     tempMat.color = color;
5     renderer.material = tempMat;
6 }
```

Code 5: Creating and using a new material.

HM_Tracker

The `HM_Tracker` script is placed on the object one wants to gather data for (see section 3.1). It will track the object's position by default and can be set up to track some other events as well (see **Code 6** on page 7 for the code that is used to track other events). It will save this data to an XML file for the session that is in progress (determined by the `HM_Control0bjectScript` script).

```

1 // Tracks the object if OnTriggerEnter is to be tracked.
2 if (TrackedEvents.Contains(HM_EventTypes.OnTriggerEnter))
3 {
4     eventsLogged.Add(new HM_Event(HM_EventTypes.OnTriggerEnter,
5         transform.position));
}
```

Code 6: Tracking an event if the event should be tracked.

5 Testing

After creating the plug-in, we decided to test it. We wanted to test how easy it was to use in other games, and we wanted to do some performance testing, to see at when the performance of the plug-in fell.

The tests were performed on a laptop with 8GB memory, Nvidia Geforce GTX 765M and Intel Core i7 processor (2.4 GHz).

5.1 Games

For the ease-of-use test, we chose three different games: Angry Bots², a 2D-platformer game³ and Hiraeth⁴.

Both Angry Bots and the 2D platformer game are demo projects that can be downloaded off of Unity's website. Hiraeth is a game we made for the Game Design-E2013 course. We tested the plug-in on all three games, to see how easy it was to set up, gather data and generate heat maps from the data.

²<https://www.assetstore.unity3d.com/#/content/12175>

³<https://www.assetstore.unity3d.com/#/content/11228>

⁴<https://github.com/esfdk/DarkForest>

5.1.1 Hiraeth

Hiraeth is a game that consists of a huge, square piece of terrain with hills and valleys, where the player can walk around and explore the world at their own leisure. Using the heat map plug-in proved to be very easy and it worked as intended. In a few minutes time, we could track data and generate heat maps. See **Figure 4** for a part of the heat map.

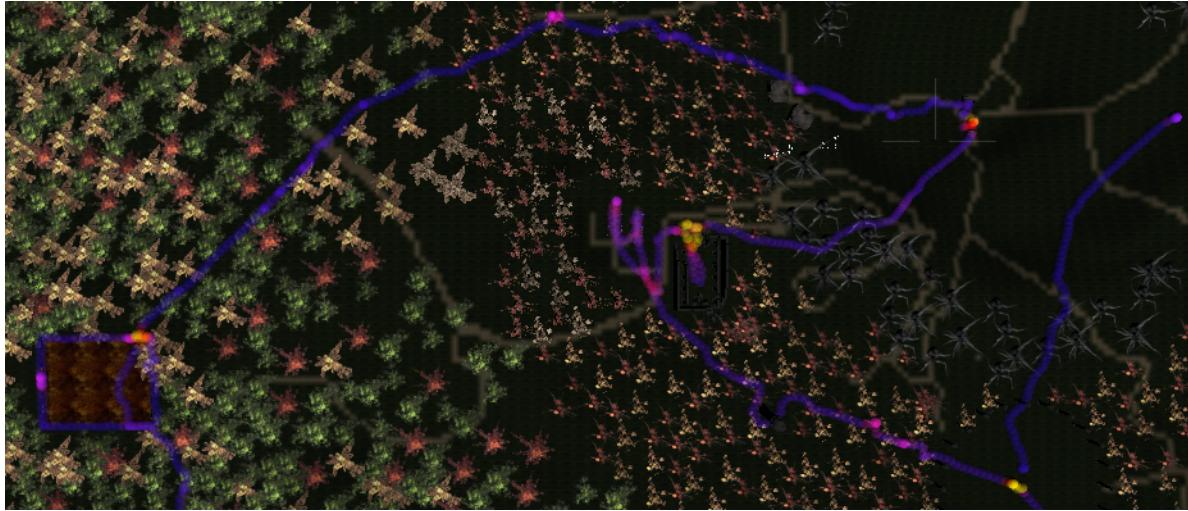


Figure 4: A heat map generated in Hiraeth.

5.1.2 Angry Bots

Angry Bots is a top-down shooter game, where the player moves through a space station, killing enemy robots in the player's way, while unlocking doors to progress further. For this game, we tested both the default BreadCrumb event, but also a custom death event for the player. Both were tracked successfully.

Generating the heat map went easy enough, but the markers were almost too transparent to be seen (see **Figure 5** for an example), so we turned the transparency down. Giving the developers an easy way to control the transparency would be a good idea for future work.



Figure 5: Derp

5.1.3 2D Platformer

Like the two other games, installing the plug-in was easy. So was gathering data and generating the heat map. However, due to the game using Unity's new 2D tools, the markers were not rendered. This is because they use the default transparent shader, which is not rendered by the 2D tools.

5.2 Performance

For the performance test, we decided to test on Hiraeth, as it was the game we were more familiar with and the one we had the most data for. We wanted to test the performance when it came to tracking a lot of objects and how fast the heat map could be generated.

5.2.1 Object tracking

For the object tracking, we chose a certain amount of elements and attached the `HM_Tracker` script to them. Then we started the game and looked for any drops in performance, be it frame rate loss or stutters when data was being saved.

We used the default values (*Position interval 1 & Save interval 10*) for all the objects, in an attempt to make as much data writing happen at the same time as possible.

Test number	Elements tracked	Impact
1	10	No noticeable impact
2	20	No noticeable impact
3	70	No noticeable impact

As can be seen from the table, we tracked up to seventy objects simultaneously, which we consider to be a lot more objects than most will need to track simultaneous. Even with that many objects, there was no impact on the performance.

From this data, we can conclude that tracking and saving of data does not need to be improved from a performance perspective.

5.2.2 Generating the heat map

For the heat map generation, we decided to take one of our large datasets and load it multiple times. Each time we would note the amount of events loaded and the time it took to load the new event and process both them and the old events.

Test number	Events loaded	Time in seconds
1	3224	37.081
2	6448	92.760
3	9672	169.625

From the table, it is easy to see that it takes a lot of time to load many events. This is because each time a heat map is generated, each marker needs to iterate over all the other markers to figure out which ones are close to it. After that, each of them needs to have their color change. This gives a run time of $O(n^2 + n)$ (or simply just $O(n^2)$), which is way too slow to be used for large amounts of data.

While 9672 events are a lot, it is very possible that some heat maps would be working with that - and higher - numbers. It is therefore an area that could do with improvements, which is discussed in section 6.2.

6 Issues

While our testing confirmed that the plug-in worked, we did run into some issues.

6.1 Data from multiple files

As the plug-in lets the user choose which file to load data from, it is possible to load data from multiple different files. It is cumbersome, however, as the user will have to choose a file, load the data, choose the next file, load data, etc. Loading data from tens of files or more will end up becoming an annoyance, rather than a helpful tool.

The solution to this, is to allow the user to select multiple files to load data from. Instead of a drop down menu, a menu that allows the user to select which files to load from would be a lot more user friendly.

6.2 Performance of heat map generation

As described in section 5.2.2, generating the heat map becomes slower the more points that are being loaded and processed, because of the $O(n^2)$ run time.

There are a couple of ways to address this issue. The first way would be to do the data processing on a separate machine. Simply choose all the data that was to be used, process it, save it to a different XML format and then load that format instead. The processing could either happen on another computer, or on a server dedicated to heat mapping data. It would still be slow, but it would not stop the developer from doing anything while the heat map was being generated.

Another way would be make a binned heat map instead of the current version. Instead of coloring areas of varying sizes, the map is split into x by y squares of a chosen size. The events are run through, and every event inside a square increases the density of that square. After all events have been run through, the squares are colored (see **Figure 6** on page 11 for an example). This way, the run time would be reduced to $O(n + m)$ ($n = \text{number of tracked events}$ and $m = \text{amount of squares}$), which is considerably faster than $O(n^2)$.

The trade-off is that a binned heat map is a top-down picture. Therefore, if the game a developer is working with has multiple floors linked together and they are on top of each other, he would have to generate a heat map for each floor. This means the run time is $O(h * (n + m))$ (where h is the amount of height maps to generate). It should never be more expensive than it currently is, however, as it would require there to be more floors than events.

6.3 Object names when tracking

For each object that is being tracked, the object will create a file named "HeatMapData;objectName;.xml" and open an XmlTextWriter to that file. This causes a problem when multiple objects have the same name, as they will overwrite previous files. For example, if ten *Enemy* objects are being tracked, only the last one to be loaded will have a file to write to, and the other nine will lack the file their writer is linked to. Thus only one object will actually be tracked.

The simple way to solve this problem, is to check if the file already exists, and in that case append a number to the end of the new file name. This would leave a lot of "HeatMapData;objectName;.xml" files in the session folder, but as the user will never have to interact with these files, it should not be an issue.

6.4 Unity 2D tool support

In version 4.3 of Unity, better 2D tools were added. As described in section 5.1.3, we found that the plug-in works when it comes to gathering data and generating a heat map from this data. However, while the heat

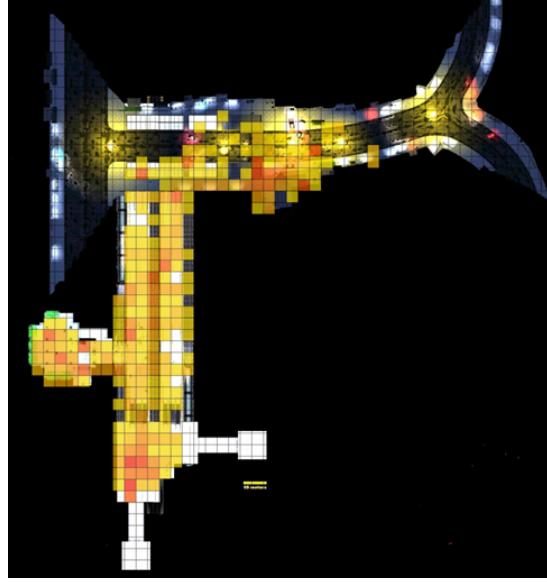


Figure 6: A binned heat map.

markers were added to the scene, they were not rendered.

One way to solve this issue, would be to create a custom shader that retains some sort of transparency, while still being possible to render.

7 Conclusion

7.1 Future work

If we were to keep developing this tool, there are a few things I would like to change.

The main thing to do, would be to let the developer choose whether he wants the 3D heat map or a binned heat map (see section 6.2 on page 10). This would allow the developer to choose between getting a very detailed heat map at the cost of time, or getting a heat map faster at the cost of perfect precision.

Apart from the other things discussed in section 6, a change to how markers are placed in the world would be in order. Currently, there is a marker for every loaded event, which is a lot of markers, which can slow down Unity. To reduce this problem, one could combine markers if they are close enough to each other. Simply remove the markers and place a new one at a point between them. This would reduce the amount of marker objects in the scene, without giving up too much precision. This should of course be something the developer can choose to use, not something they are forced to do.