Unity Heat Map Plug-in

Game Engines E2013

IT-University of Copenhagen

Jacob Claudius Grooss, jcgr@itu.dk

December 11th, 2013

Contents

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 FPS[1] 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?

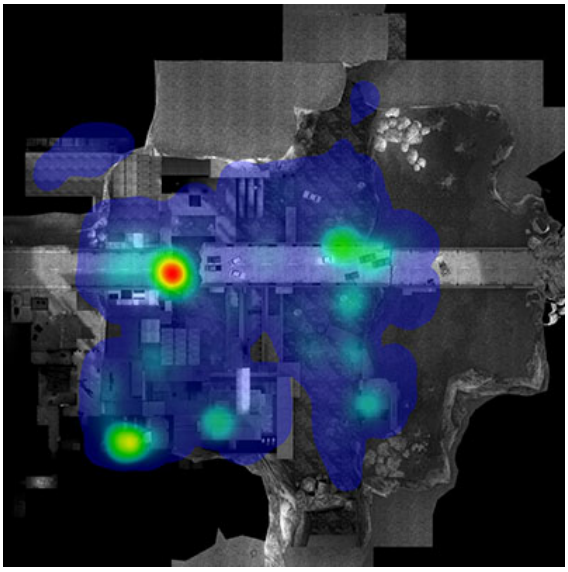

Figure 1: *A heat map representing deaths in a level in Half-Life 2*

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

[1] First 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 things 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 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.

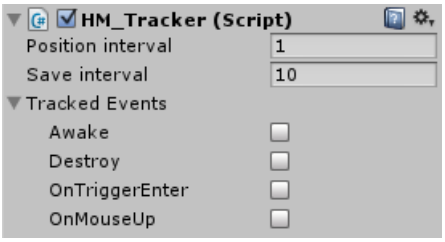

Figure 2: *The tracker options.*

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 OnMouseUp.

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.

OnMouseUp 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.

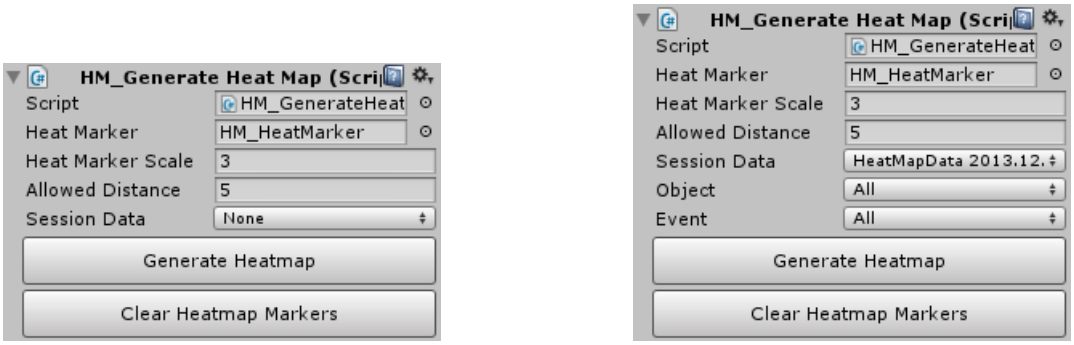

Figure 3: *The options for generating heat map. Left picture is without any data selected, the right is with data selected.*

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 `HSBColor`, 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 is the one responsible for generating the heat map (see section 3.2 on page 4 for how to use it).

`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
4         if(!Directory.Exists(directory))
5         {
6                 Directory.CreateDirectory(directory);
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 +
              "/" + actualEvent);
2
3         // Iterate over all the event nodes and create markers for them.
4         foreach(XmlNode node in xnlNodes)
5         {
6                 var x = float.Parse(node.Attributes["x"].Value);
7                 var y = float.Parse(node.Attributes["y"].Value);
8                 var z = float.Parse(node.Attributes["z"].Value);
9                 var hm = (GameObject) Instantiate(HeatMarker, new Vector3(x, y, z),
                      new Quaternion(0,0,0,0));
10                hm.transform.localScale = new Vector3(HeatMarkerScale,
                      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 setion 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_ControlObjectScript` script).

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

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 Bots[2], a 2D-platformer game[3] and Hiraeth[4].

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.

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.

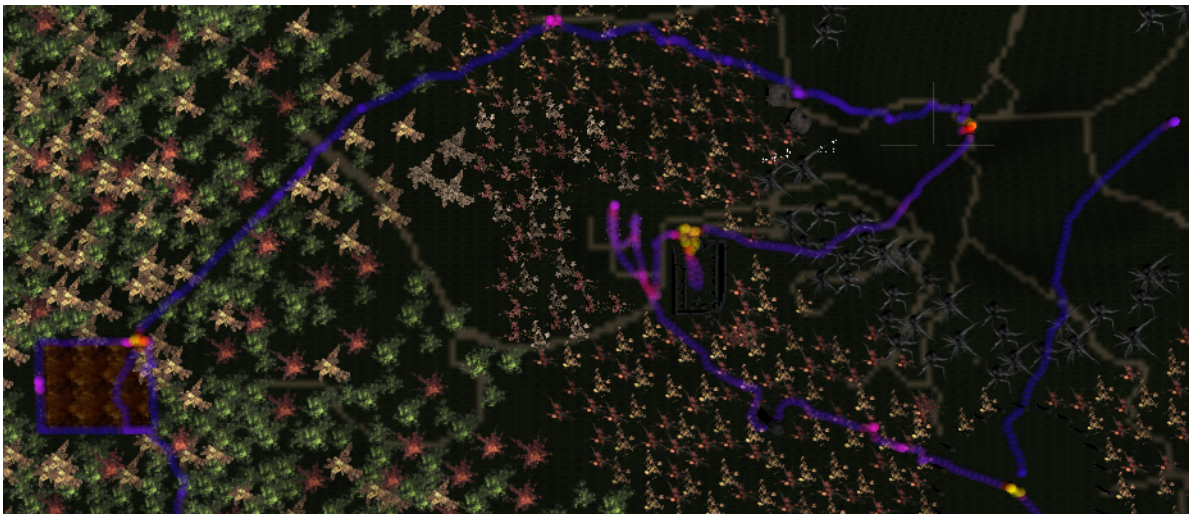

Figure 4: *A heat map generated in Hiraeth.*

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.

[2]https://www.assetstore.unity3d.com/#/content/12175
[3]https://www.assetstore.unity3d.com/#/content/11228
[4]https://github.com/esfdk/DarkForest

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.

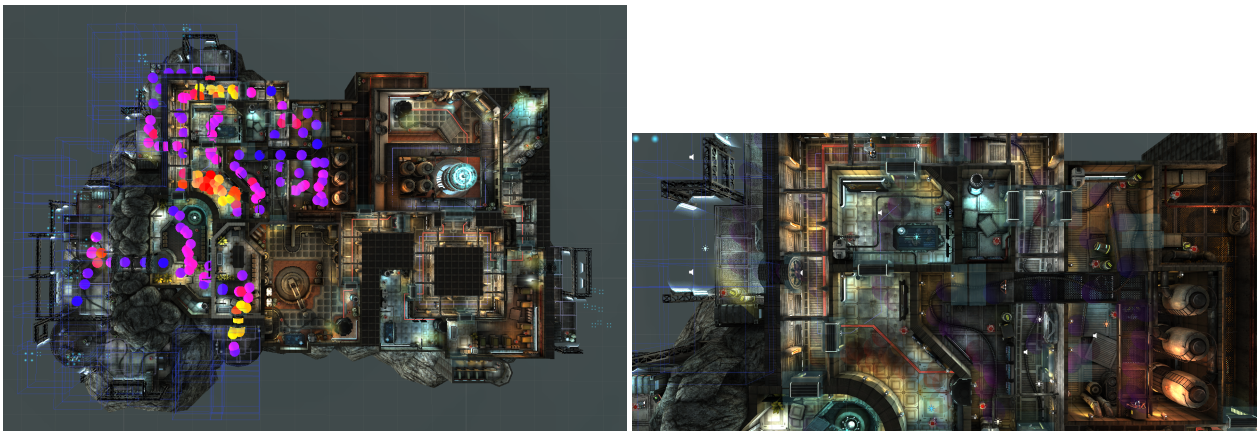

Figure 5: *Angry Bots heat map with different levels of transparency.*

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	3224	35.106
3	6448	61.844
4	6448	86.323
5	9672	169.625
6	9672	129.206
7	12896	238.759
8	12896	218.894
9	16120	311.298
10	16120	325.748
11	19344	443.752
12	22568	581.286

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 22568 events are a lot, it is 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 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 12 for an example). This way, the run time would be reduced to $O(n + m)$ (n = *number of tracked events* and m = *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 exits, and in that case append a number to the end of the new file name. This would leave a lot of "HeatMapData¡objectName¿¡number¿.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

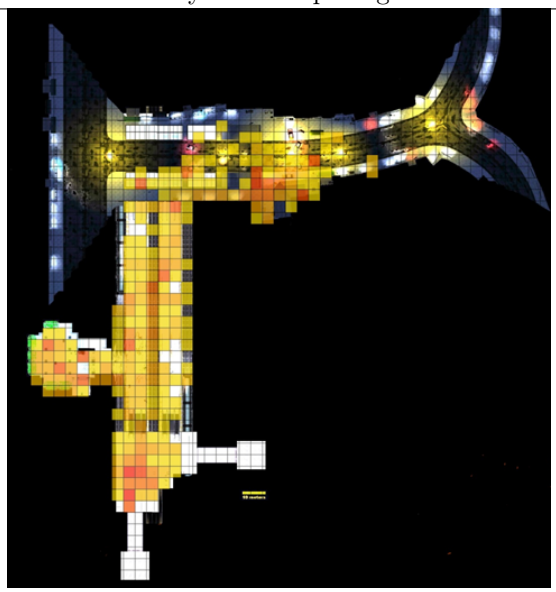

Figure 6: *A binned heat map.*

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

For this project, Jakob Melnyk and I decided to write a tool that could generate heat maps for a game. After some talking back and forth, we decided to write the tool for Unity, as we felt it would be more useful. We then decided on some features we wanted the tool to have and started writing it. Near the end of the process, we ran tests on the tool to confirm whether it worked or not.

From the tests - and the fact that the tool fulfills the feature requirement we set - we can conclude that the tool is capable of generating heat maps of decent quality for games made in Unity. It allows the user to specify which objects to track, and which events to track for the object. The tracked data is automatically saved to a file, which can be loaded by the file again. This data can be filtered by session, object and event and used to generate a 3D heat map overlay for the game.

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 11). 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.

Game Engines

MGAE-E2013

2D Platformer Engine

IT-University of Copenhagen

Jacob Claudius Grooss, jcgr@itu.dk

May 22, 2013

1 Introducton

This report has been written as part of a project on the Game Engines E2013 (MGAE-E2013) course on the Games course at the IT-University of Copenhagen.

For this assignment I have created a game engine for a 2D platformer game, written in C++ with the use of the SDL framework[1]. I have worked, and shared ideas, with Jakob Melnyk (jmel).

2 Features

The engine is intended to support simple 2D platform games built in tiles, where jumping and avoiding enemies while trying to reach a goal area are the key points. The engine contains the following features:

- A player that has the ability to walk and jump.
- Enemies that fly or walk. In the case of the walking enemy, it is also affected by gravity.
- Collision detection, both between entities (player/enemies) and the walls, and between the player and the enemies.
- A side-scrolling camera that follows the player.
- Animations for entities.

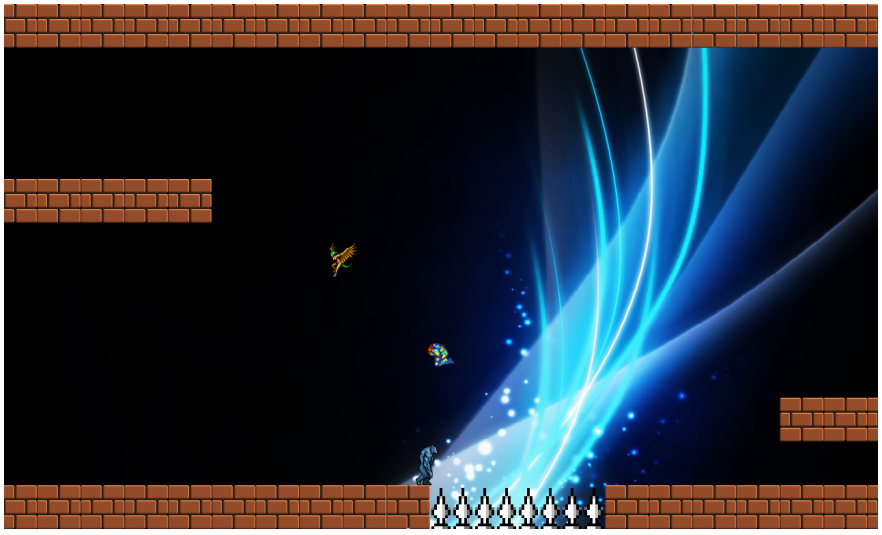

Figure 1: A picture of the game in progress.

3 Overall Structure

In this section, I will go over the structure of the engine, which consists of the following classes: A map, a player, enemies, a level, a window to display everything to and a class for shared functions.

[1] http://www.libsdl.org/

Map

The **map** class is a representation of the world the player is traversing. It contains information about the layout of the world, such as where solid blocks and spikes are found, where the goal is, where the player spawns and where enemies spawn. The information about the world is loaded from a .txt file (see section 6.1 for an example and explanation of the data format).

Player

The person playing the game needs to be able to interact with it, which is the purpose of the **player** class. It represents the player character, which can move around the map and collide with other objects. Collision results in different events depending on what is hit (stopping if a wall is hit, dying if something hostile is collided with, etc.).

Enemies

Obstacles in a 2D platformer is important and some kind of enemies are usually used for this purpose. In the engine, enemies are represented by the **enemy** class, which is a very basic enemy. It contains very basic behaviour, but it can be extended by other classes to create specific enemies with their own behaviour. This makes it very easy to implement new enemies.

Level

The **level** class represents an entire level in the engine. It contains a map, a list of the enemies in the map and the player character.
The level handles input from the keyboard tells the player to move depending on the input. The level also updates and moves the enemies according to their behaviour. It also takes care of telling the window class where to draw everything.

Window

The **window** class is the one that handles any form for drawing to the screen. It initializes SDL, creates a window for it to draw on and helps with loading and drawing of images in an easy way. It is used behind the scenes.

HelperClass

As there are some things that are used by multiple classes (for example the amount of collision points for entities), having such information gathered in one place makes it easier to keep track of. That is the purpose of the **helperclass** class. It contains values and functions used by multiple other classes, so they do not have to be defined a lot of different places.

These classes are the main ones that make up the engine. Used together they can produce a simple 2D platformer game, in which the player's objective is to get from one point to another.

4 Improvements

Currently, the engine works to a certain degree. It supports simple 2D platformer games, where the player has to dodge enemies and deadly obstacles in order to reach a goal. But that is pretty much all it supports. There is no fighting, interacting with objects, acquiring new abilities or anything along those lines. Implementing such things would be one of the main focuses if improvements were to be made.

When it comes to implementing new things, enemies work fairly well. One can extend the **Enemy** class to create new kinds of enemies with different behaviours, but loading them into a level from a map file is not easy. The reason is that the map uses a very basic data format, which is not easy to extend. Therefore, changing the data format of the map files would be something to look into.

Introducing an actual camera class would be a welcome change as well. In the current implementation, there is no actual camera. Things are just drawn to the screen in relation to where the player character currently is. A separate camera would be easier for the user of the engine to work with, as the user would have more control over how it works.

Another thing that could be improved upon is the performance on fast vs. slow computers. As it is, there is nothing that takes framerate into account, and the engine will therefore run slower on older/slower computers, which worsens the experience for people with slow computers.

These are just a few of the possible improvements. Each of them would take some work to get done, but each would make the engine a lot better.

5 Conclusion

The engine I have written allows for users to create a very simple 2D platformer game. The engine is fairly simple to use, but there is very little customization, The games that can be created with it revolves around the player navigating his/her was from the start to the goal, while avoiding enemies and lethal obstacles.

6 Appendix

6.1 Map data file and layout

The map uses the following data format: The first line determines the amount of rows in the map, the second determines the amount of columns. The lines after represents the layout of the map, with the numbers meaning the following things:

- **0**: An empty space.
- **1**: A solid block.
- **2 & 3**: 2 and 3 both represent the goal, but there is a difference: 2 determines the place where the goal texture is drawn and is used for checking for collision with the goal, where 3 is only used for checking for collision with the goal.
  This two-part design is used because drawing the goal texture multiple places looks strange, but collision might be needed for multiple tiles. 3 was introduced to handle that.
- **4**: Spikes.
- **5**: The spawn point of one kind of enemy.
- **6**: The spawn point of another kind of enemy.
- **9**: The spawn point the player.

```
1  12
2  40
3  1111111111111111111111111111111111111111
4  1000000000000000000000000000000000000001
5  1000006000000000000000000000000000000001
6  1000000000000000000000000000000000000001
7  1111111110000000000000000000000000000001
8  1000000000000000000000000000000000000001
9  1000000000005000000000000000000000000001
10 1000000000000000000000000000000000000001
11 1000000000000000000000000000000000000001
12 1000000000000000000001111000000000300001
13 1090000000000000000000000000000000200001
14 1111111111111144441111111111111111111111
```

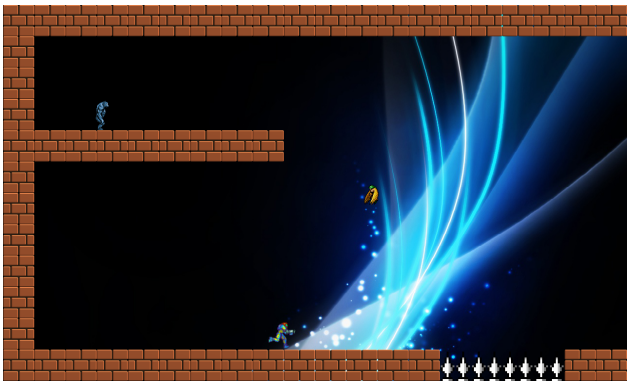

Figure 2: A picture of the game in progress.

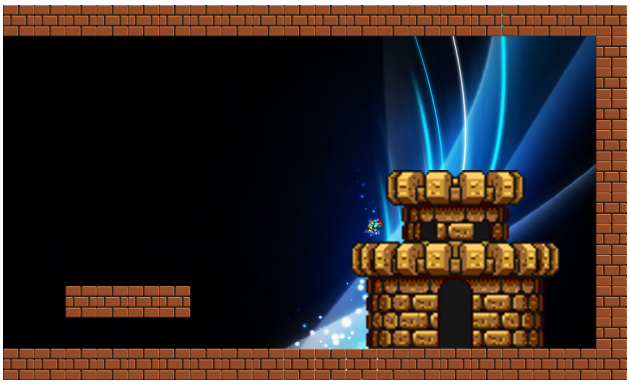

Figure 3: The player about to get to the goal.

Figure 4: The screen shown after completing a level.

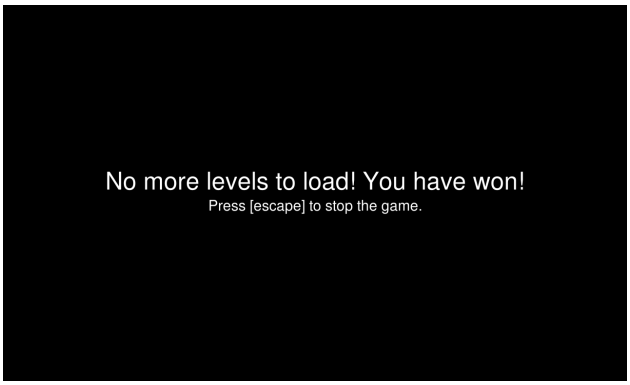

Figure 5: The screen shown when all the levels have been completed.

Game Engines

MGAE-E2013

Wireframe Rendering

IT-University of Copenhagen

Jacob Claudius Grooss, jcgr@itu.dk

May 22, 2013

1 Introducton

This report has been written as part of a project on the Game Engines E2013 (MGAE-E2013) course on the Games course at the IT-University of Copenhagen.

For the project, I have written a program in C# (with the use of Windows Forms) that can render simple wireframe models. For the calculations, I followed the the "Overview of transforms used in rendering"[1] tutorial by Mark Nelson. The program has been written in C#. I have worked, and shared ideas, with Jakob Melnyk (jmel).

2 Features

In this section I will describe the two main features of the program: Rendering and camera movement.

2.1 Rendering

The program utilises a list of triangles created from vertices for rendering. The program calculates the location of each triangle on the screen based on the camera's variables and draws them to the screen.

2.2 Camera Movement

The camera can be moved around to display the wireframe model from different angles and positions. The controls for moving it are:

- 'a' and 'd' moves the camera on the x-axis.
- 'w' and 's' moves the camera on the y-axis.
- 'q' and 'e' moves the camera on the z-axis.
- 'j' and 'l' moves the look point of the camera on the x-axis.
- 'i' and 'k' moves the look point of the camera on the y-axis.
- 'u' and 'o' moves the look point of the camera on the z-axis.

3 Overview

In this section I will be giving an overview of the program. I will cover the classes of the program, how the rendering and math works and in which parts of the program the math is.

3.1 Classes

When it comes to rendering wireframe models (or entire 3D models), things such as vertices and matrices are used. I have decided to map these things to classes, along with a few others needed to run the program.

Matrix, Vertex, Triangle and Vector

These four classes are computer representations of their mathematical versions.

- `Matrix` represents a matrix of varying sizes and holds floating-point values.

[1] https://blog.itu.dk/MGAE-E2013/files/2013/09/transforms.pdf

- `Vertex` represents vertices, but has four coordinate points (x, y, z and w). The fourth, w, is a dummy value so it matches the four-dimensional matrices that are used. It also has a point (an x- and y-value) that represents its position on the screen.
- `Triangle` represents a triangle in 3D space and contains the three vertices that determines the triangle's corners.
- `Vector` represents a 3D vector has three coordinate points (x, y and z).

Camera

The `Camera` class represents the camera in the model view space. It is what allows the user to see the wireframe models. It contains variables that determines its position, where it is looking, its height, width, field of view and aspect ratio.

WireframeRenderer and Program

The `Program` class starts the application. It creates a new instance of the `WireframeRenderer`, which is a subtype of WindowsForms. `WireframeRenderer` creates a window for drawing, initializes the camera and is responsible for accepting user input (and handling it correctly), updating the triangles and drawing to the screen.

3.2 Rendering

When the program is started, `WireframeRenderer` loads four `Triangle`s. These triangles represent a model of a pyramid, which is what the program renders.

When the model is to be drawn to the screen, it tells the camera to calculate its transforms. After the calculations, each triangle tells its vertices to update their screenpoint, which they do through the use of the camera transformations. Lastly, the triangles are drawn to the screen.

3.3 Math

I decided to move the calculations to the classes they were related to.

The `Matrix` class contains a simple method for multiplying matrices with each other.

The `Camera` class contains the methods needed to calculate the three camera transforms described in the tutorial PDF (location-, look- and perspective-transforms). It also contains a method for calculating the combination of these three transformations. It was implemented to make it less code-heavy to get the combined transform, for example for use in updating the screenpoints of the vertices.

The calculation of the screenpoints of the vertices happens across multiple classes. `WireframeRenderer` iterates over all triangles in the model, and tell each to update its vertices. The triangle then tells each of its own vertices to update their screenpoint. The vertices themselves handles the actual calculation of screenpoints.

It should be noted that I have omitted a part of the pseudocode that describes how to calculate the screenpoints. I decided not to ignore a point if it was outside the screen. The reason is, that while the point might not be on the screen, the line between points may. Skipping a point will therefore result in missing lines of the model.

4 Problems

While the program can do simple rendering, there are a few problems. One of them arises when the camera is moved through the model and no longer looks at it. The results for the screenpoints will overflow and cause the program to crash. This could be avoided by updating only the triangles the camera is looking at.

Another problem happens when the camera is turned. At some point, the transforms will end up putting one of the closer to the oppesite side of the screen from where the rest of the model is, which means the triangle will stretch across the screen. A check to see where the rest of the triangle is would help in getting rid of this problem.

A smaller problem is performance-wise. Each triangle has three vertices that needs to have their screenpoints calculated. But some of these vertices are the same for the triangles (for example the top of the pyramid). This means that some vertices may be updated multiple times, resulting in a lot of extra calculations. This could be avoided by letting each triangle hold a reference to the vertices, instead of having them hold instances of the same vertex.

5 Conclusion

The program I have written allows for rendering of simple wireframe models. Furthermore, the program allows the user to move the camera around in the model view space and inspect the model from different angles. However, large models/scenes would be problematic to render due to the lack of optimization during the calculations.

5.1 Future works

There are plenty of things that could be changed in the future.

As it is now, there are a lot of unnecessary calculations, and there are cases where the calculations result in the program crashing. This could be avoided by proper optimizing of the calculations and checking if the calculations are within the expected scope.

The camera could also use some changing. The main thing to focus on would be to allow the mouse to control the camera, instead of having to use the keyboard. The keyboard is a clunky way of controlling the camera, and it not very precise. Changing how the camera rotates would also be part of this work.

6 Appendix

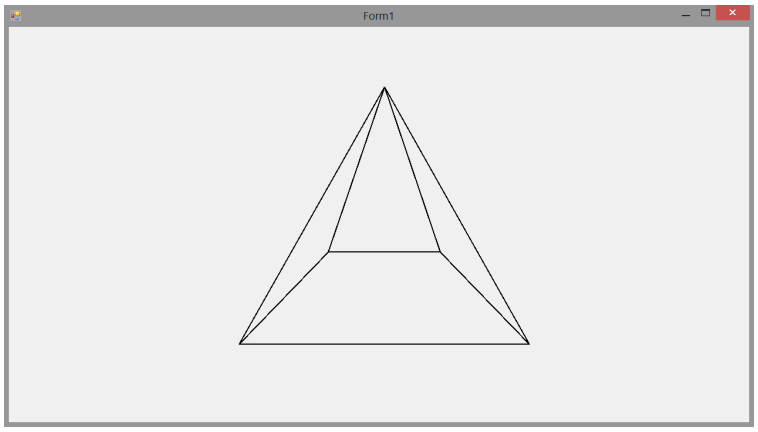

Figure 1: The program when it starts.

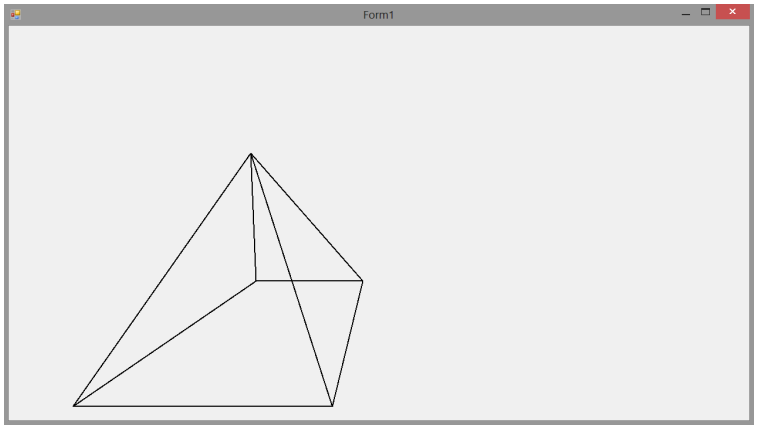

Figure 2: The camera has been moved.

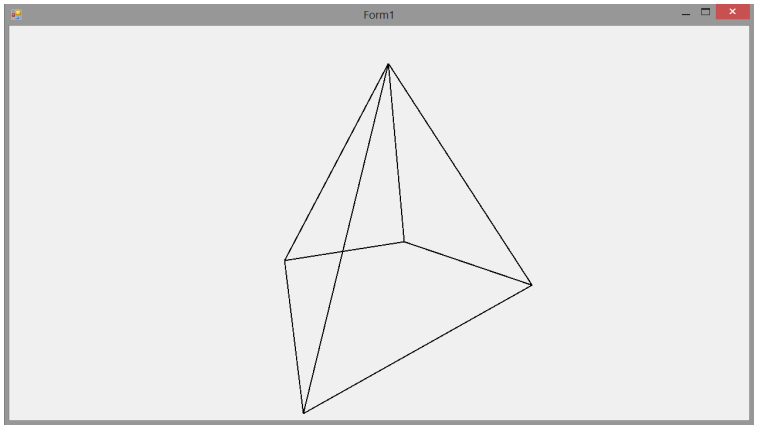

Figure 3: The camera has been moved and turned.

Game Engines

MGAE-E2013

Pathfinding

IT-University of Copenhagen

Jacob Claudius Grooss, jcgr@itu.dk

May 22, 2013

1 Introducton

This report has been written as part of a project on the Game Engines E2013 (MGAE-E2013) course on the Games course at the IT-University of Copenhagen.

For the project, I have written a program in C# that visualizes one agent chasing another through the use of the A* algorithm. I have worked, and shared ideas, with Jakob Melnyk (jmel).

2 Features

In this section I will describe the features of the program: Pathfinding, visualization and user input.

2.1 Pathfinding

The program uses the A*[1] algorithm to simulate two agents, one chasing the other. The A* algorithm allows the chasing agent to find the shortest path to his target while avoiding obstacles and, depending on his speed, catch the fleeing agent.

2.2 Visualization

The map the agents are traversing is visualized through simple ASCII art, as seen in figure 1. Agent A is represented by the capital `A`, agent B by the capital `B` and the walls by capital `X`. The paths the two agents take are represented by non-capital `a` and `b`.

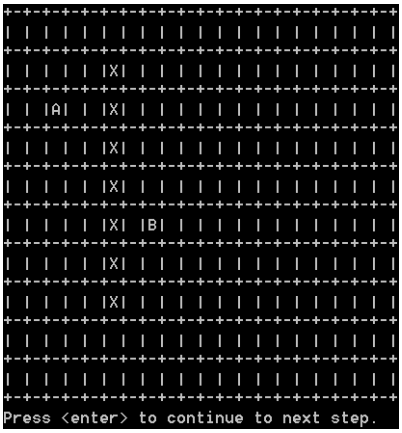

Figure 1: The initial visualization.

2.3 User input

The program accepts very simple user input. When the program is started, it lets the user specify the start position and speed of each of the agents. After that, it waits at every step for the user to press the `enter` button, upon which the simulation continues.

[1] http://en.wikipedia.org/wiki/A*_search_algorithm

3 Overview

In this section I will be giving an overview of the program. I will cover the classes of the program and how the pathfinding is done.

3.1 Classes

The program contains five classes. They are `Agent`, `Node`, `Map`, `AStar` and `Program`.

3.1.1 Agent

The `Agent` class represents the agents running around. It contains variables that specify the agent's position, speed and the path the agent currently is moving. The class has two methods, one for chasing another agent and one for fleeing from a given agent. The difference between the two is describes in section 3.2.

3.1.2 Node

The `Node` class represents the nodes the agents are walking on. Each node contains its x- and y-coordinates (used when the position of a node needs to be found without asking the map), a value that indicates if the node can be paased or not and values that are used by the `AStar` class for determining the best path to another node. It has methods for comparison with other nodes and a method for determining the distance to another node.

3.1.3 Map

The `Map` class represents the map through a 2D array of nodes. The class furthermore has methods for determining if a node (or x/y position) is within the bounds of the map, a method for drawing the map in ASCII art and a method for finding neighbours to a given node.

3.1.4 AStar

The `AStar` class contains the pathfinding algorithm, along with the method used to reconstruct the path when a route has been found. It is based on the pseudocode from the Wikipedia article about A*, but has been modified to accept a map as input as well, as it uses the map to find the neighbour nodes.

3.1.5 Program

The `Program` class starts the application and loads both the map and the agents. After loading it enters a loop, in which the agents are told to chase or flee from the other agent. When the loop ends, the program writes at which position the fleeing agent is caught, after which it terminates.

3.2 Pathfinding

Pathfinding is done in the following two ways:

- **Chasing** - When an agent is chasing another agent, it uses the basic A* algoritm to find the shortest route to the target agent.
- **Fleeing** - When an agent is fleeing, it goes through its neighbour nodes, looking for one that puts it further away from the chasing agent. It chooses the one that gives the most distance and then it checks the new position's neighbours the same. This iteration happens ten times. After the ten iterations, the agent uses the A* algorithm to find the shortest route from its current position to the position it found to be furthest away from the chasing agent.

4 Problems

A major problem in the program lies with the fleeing agent. Most of the time it will act in a smart way, until the chasing agent is on the same row or column as the fleeing agent. At this point, the fleeing agent will prefer moving either up or left (depending on if they are on the same row or column) until it reaches the edge of the map. Here it will move away from the edge, only to go back to where it came from.

This issue only occours when the fleeing agent has a speed of 1. It could be prevented by a better prediction of where a certain node will lead it, instead of blindly going for the one that puts it the furthest from the chasing agent.

5 Conclusion

The program I have written simulates one agent chasing another through the use of the A* pathfinding algorithm. It allows the user to specify the start position and speed of the two agents and will visualize each step of the chase.

5.1 Future works

While the program can do simple pathfinding/visualization, there are a lot of things that could be changed:

- **GUI** - The GUI could be improved tenfold (or more). Currently it just consists of printing to the console. While it gives a basic view of what is going on, it is by no means pretty or user-friendly. Changing it to include graphics would be a huge improvement and would also make user interaction (see next point) a lot easier.
- **User Interaction** - As it is, the user is only allowed to interact with the program in a very limited way: At first by selecting the start position and speed of the agents, then by controlling the visualization by pressing `enter`. Letting the user open/close nodes or changing the position/speed of the agents while the program is running would make it more interesting.
- **AI** - When it comes to movement, the chasing agent is doing fine, but the fleeing agent has some issues (as described in chapter 4). The method for finding a suitable path away from the chasing agent would be an improvement worth spending time on developing.
  On the same note, implementing states for the agents ("idle", "chasing", "fleeing", etc.) would also be worth looking into, as it would better simulate how beings act.
- **Terrain types** - All nodes (excluding the closed ones) are treated equally during pathfinding. Introducing different types of "terrain" (water, forest, plains, mountains, etc.) would make the simulation more interesting, as the agents would have to take the terrain types into account. This would require the use of a graphics-based UI instead of the current text-based one, but would be well worth the time.

6 Appendix

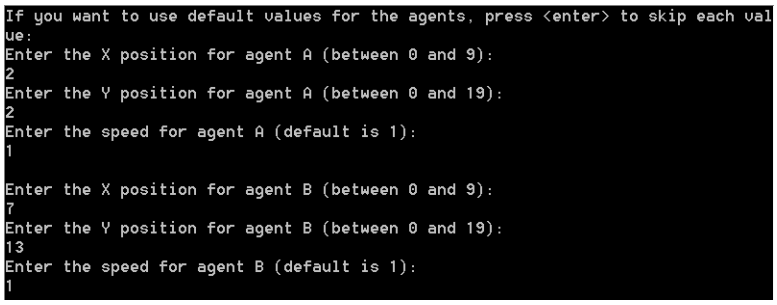

Figure 2: Choosing the start position and speed of the agents.

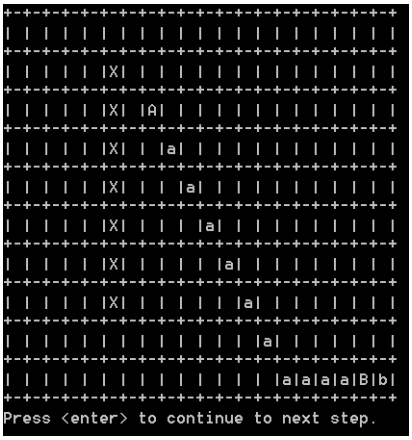

Figure 3: A few steps into the simulation.

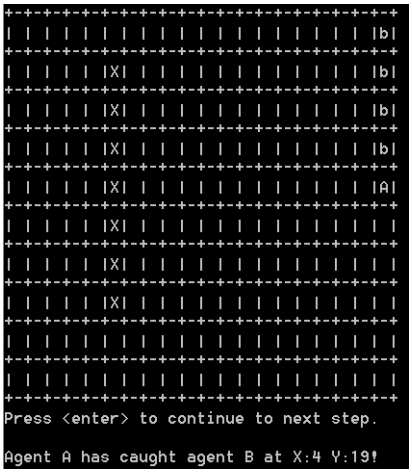

Figure 4: The end of the simulation.