Fakultät für Informatik
Technische Universität München

Bachelors's Thesis in Informatik: Games Engineering

Gameplay Analysis of Competitive Games to Support the Game Design Process

Luca Ian Türk

Fakultät für Informatik
Technische Universität München

Bachelors's Thesis in Informatik: Games Engineering

Gameplay Analysis of Competitive Games to Support the Game Design Process

Gameplay Analyse von kompetitiven Spielen zur Unterstützung des Game Design Prozesses

Author:	Luca Ian Türk
Supervisor:	Prof. Gudrun Johanna Klinker, Ph.D.
Advisors:	Daniel Dyrda, M.Sc.
Submission Date:	September 15, 2020

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich diese Bachelors's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I confirm that this bachelors's thesis is my own work and I have documented all sources and material used.

Munich, September 15, 2020

LUCA IAN TÜRK

Abstract

Competitive gaming is a growing market with nearly 500 million fans worldwide. [32] With professional gaming on the rise, developers are under increased pressure to create a fair and balanced gameplay experience.

The goal of this thesis is the creation of analysis tools, which support developers of competitive games during the game design process. Although various genres are represented in competitive gaming, the focus was laid on first person shooter games, specifically the Counter Strike series of games. Based on various map design guides, a number of analysis tasks were defined, notably the identification of meeting points and the evaluation of visibility between map areas.

For this purpose, a general framework for the analysis of game maps and visualization of analysis data for use in the Unity Editor is implemented. A uniform grid representation of the walkable map areas, which can be generated for arbitrary maps, forms the basis of this framework. Additional tools allow the definition of map areas and locations.

Using a Monte Carlo approach, the mutual visibility between map areas is evaluated. The evaluation is shown to predict possible angles of engagement on the Counter Strike map *de_dust2*, as well as reveal possibly unintended sight lines between map areas. Utilizing the NavMesh and pathfinding components provided by Unity, a distance evaluation tool was implemented. An evaluation of meeting points for *de_dust2* using this tool closely matched known meeting points for the map.

Contents

1 Introduction

In July of 2019, an unusual view could be seen in Cologne's Lanxess-Arena. The hall, usually hosting the local ice hockey and basketball clubs, is filled to the brim. In place of hockey goals or basketball hoops, a stage is put up. [39] On it, a row of computers is backed by giant screens. As a group emerges, people hold out their hands for a chance to high-five the players. The event is ESL One, a Counter Strike : Global Offensive tournament with a 300.000 $ price pool, featuring eleven teams from around the world.

So called e-sports have been steadily increasing their viewership numbers over the last decade. In 2020 nearly 500 million people followed such competitions at least occasionally [32], rivaling some of the top ten traditional sports in viewership. [33]

With professional e-sports competitions on the rise, there is an increased need for fairness in video games.

In traditional sports, fairness is often achieved through symmetry, both in objective and playing field. Video games on the other hand thrive for variety, often featuring a multitude of maps and asymmetric objectives.

Designing fair maps for such games is not an easy task, and the process is often based on the intuition of the map designer and extensive play testing. Many gameplay-relevant facets of a maps design could, however, be evaluated automatically during the map design process.

In this thesis a general framework for the automatic analysis of game maps will be proposed, and a number of analysis tools relevant to the gameplay of the Counter Strike series of games will be implemented.

To support designers during the map creation process, the presented tools are integrated into the editor of Unity Engine, the top game engine in terms of market share with over 50% of games being powered by Unity in 2019. [35]

Figure 1.1: 15.000 fans follow a Counter Strike: Global Offensive tournament in Cologne's Lanxess-Arena, photo by Henning Kaiser [23]

2 Interesting Map Features

In the following chapter, a number of interesting map features will be presented that form the basis for the performed analysis.

2.1 First Person Shooters

First person shooters (FPS) are action games played in a first person perspective that allow players to engage in virtual combat. Engagements between players are usually fought using guns or other ranged weapons.

The skill in FPS games is derived from the players ability to effectively navigate the virtual world and overcome enemy players in firefights, requiring them to react quicker and be more accurate than their opponent.

The movement and gunplay are the most important pillars of an FPS games design, and in fact, many players choose to partake in so called death matches that feature no additional objectives.

Over the years, a number of common game modes have been introduced that feature more objectives and increase the tactical complexity of FPS matches. Some examples are *capture the flag* (CTF) game modes that require players to steal a flag out of the enemy teams base or *domination* style game modes that feature specific map areas which need to be captured and held by the teams.

2.1.1 Counter Strike

Counter Strike (CS) is a tactical multiplayer shooter series developed by Valve. The first game of the same name was developed as a modification to Valves successful shooter Half-Life. A first version was available for play in 1999, with a full release following in 2000 after being acquired by Valve. [8]

Counter Strike models a conflict between terrorist (T) and counter terrorist (CT) factions taking after real world terror organizations and police forces. In accordance with its setting, the game features multiple modes such as hostage and bombing scenarios. [8]

While many iterations have since been released, CS stays close to its roots, adding only minor changes to the formula. Subsequently is has been continuously played competitively since 2001. The most recent iteration Counter Strike: Global Offensive (CS:GO) was released in 2012. [9]

Maps for the Counter Strike game mode bomb defusal will be used throughout this thesis. Additionally the map design principles underlying the implemented tools are based on this game mode.

Figure 2.1: Impressions from Counter Strike: Global Offensive. Advance of T side players and planting of the bomb on the map *de_inferno*.

Bomb Defusal

In bomb defusal, players of the terrorist side try to plant a bomb at one of two bomb sites on the map. The counter-terrorist team on the other hand defends the bomb sites and, in case the bomb is planted, try to defuse it before is blows up on a 40 second timer. [6]

The game mode allows players to make meaningful tactical choices. At the outset of a round, both teams have to choose which bomb site to attack or defend, or to split up the team to cover both. If players miscalculate they need to hurry to the other site in a race against time.

Contrary to most arena shooters common at the time of its inception, Counter Strike features no respawn mechanic. If players get killed they are out for the rest of the round, further raising the stakes.

Bomb defusal is currently the only mode played in competitive Counter Strike e-sport events.[7]

2.2 Interesting Map Features for Analysis

Choke Points

Choke points are the areas where the attacking team will encounter the defending team and, therefore, where most of the action will take place. According to a guide by Alex Galuzin, a map designer and video tutorial creator, successful choke point design in CS depends on multiple principles.

First, it needs to be placed in such a way that it will be reached at almost the same time by both teams. The defending team should reach the choke point slightly ahead of the attacking team, such that a defense can be set up. In no circumstance should the attacking team be able to rush through the choke point while the defending team has not reached it yet.

The choke point should narrow down the map in order to force a meeting of the teams, while the attacking team should be afforded some options, for example by providing multiple access points. Defenders should then be able to view all entrances from a single viewpoint located on the choke point.

Lastly the sum of choke points on any given map should facilitate different play styles,

Figure 2.2: A defenders perspective of choke point *banana* on the map *de_inferno* from Counter Strike: Global Offensive

such as sniping, close quarter combat or stealth. [16]

One example of such a choke point on map *de_inferno* from Counter Strike: Global Offensive can be seen in figure 2.2. The area named *banana* due to its curved shape visibly narrows down the map, resembling a tight alleyway. Both the defending and attacking team are provided with cover, in the form of barrels and other objects, as well as nooks in the alleyway.

Meeting Points

Meeting points are then the the areas where players of the opposing teams can first meet, given that they are advancing as fast as possible.

Rotation Time

Rotation time in Counter Strike refers to the time needed to get from one bomb site to the other. If CT players set up their defense at the wrong bomb site, they need to be able to get the other site before the 40 second timer runs out. Additionally some leeway should be given, such that the defense set up be the T team can be overwhelmed. A community map design guide published on the steam forums suggest a shortest rotation time in the range of 10 to 15 seconds. [14]

Visibility

Visibility, or a sightline, exists between two points on the map if players located at these points are able to see, and subsequently engage, each other. The area visible to a player is then also an area controlled by that player, similar to chess pieces controlling the squares they can take on.

Figure 2.3: Example of a pixel angle on the map *de_mirage* from Counter Strike: Global Offensive [14]

According to the before mentioned guide, long sightlines should be avoided in Counter Strike, as an unfair advantage is given to players equipped with sniper rifles. Other problematic sightlines are tight angles and pixel angles. These are, often unintended, sightlines between areas that are unintuitive to players and give advantage to those willing to abuse them. An example of such a pixel angle provided in the guide can be seen in figure 2.3.[14]

2.3 Main Tasks

The main tasks performed by the implemented tools are then the following.

2.3.1 General Framework

The project provides a general framework for the implementation of analysis tools, notably a grid mesh structure that supports the sampling of uniformly distributed possible player positions and the rendering of analysis data. Additionally, tools are implemented that allow map designers to designate areas and locations on the map which can be used in analysis tools.

2.3.2 Path Evaluation

The tools support the evaluation of single paths, as well as the distance from a number of spawn points to every point of the map. Utilizing these tools, walk distances between areas can be evaluated to fine tune timing and meeting points can be visualized to verify choke point placement.

2.3.3 Visibility Evaluation

The regions visible to players at any map area can be visualized. Uses for this tool are found in the evaluation of the entry point visibility, as well as the play style supported on a choke points. Additionally, the tool can be used to identify long or unintended sightlines between areas.

3 Grid Mesh Generation

This chapter will describe how a uniform grid mesh can be generated on the basis of the Unity Navigation Mesh (NavMesh). The resulting GridMesh forms the basis for the following evaluation tools and can render data directly onto the maps surface.

3.1 NavMesh

The NavMesh concept was popularized in video game programming by Greg Nook in the early 2000s. The method creates a simplified mesh representation of the walkable surface area of a map that can be used to perform pathfinding on a greatly reduced data set compared to the original level geometry. [31] Unity Engine is shipped with a NavMesh implementation that supports the generation of NavMeshes in the Editor, as well as a pathfinding implementation.

3.2 Generating the GridMesh

As evaluations and data visualization should be performed with equal detail across the map, a uniform grid needs to be formed representing possible player positions on the map. The NavMesh is a good basis for this mesh, as the walkable area of a map includes most possible player positions while excluding unnecessary geometry.

Initially the algorithm is described for the single layer case, meaning that the map cannot feature any vertically overlapping map areas, and is then extended the the general case.

3.2.1 Calculating the Optimal Resolution

To ensure a uniform distance of vertices on the xz plane, the ratio of the grids resolution needs to match the ratio of the NavMesh's dimensions on that plane.

$$\frac{dim.x}{dim.z} = \frac{width}{depth}$$

With:

$$dim = \begin{pmatrix} |max(\{v_x | v \in V_N\}) - min(\{v_x | v \in V_N\})| \\ |max(\{v_y | v \in V_N\}) - min(\{v_y | v \in V_N\})| \\ |max(\{v_z | v \in V_N\}) - min(\{v_z | v \in V_N\})| \end{pmatrix}, \text{ where } V_N \text{ the NavMesh vertices}$$

The maximum number of vertices per vertical layer n is defined by the user.

With $width * depth = n$, the resolution can then be calculated:

Input:
V: a two dimensional array of vectors of sizes $width$ and $depth$
max, min: Vectors of minimum and maximum values of vertex positions in V_N

for $i \in [0, width - 1]$ **do**
  for $j \in [0, depth - 1]$ **do**
    $scanhead \leftarrow \begin{pmatrix} Lerp(\, min_x,\, max_x,\, i/(width-1)\,) \\ max_y + c \\ Lerp(\, min_z,\, max_z,\, j/(depth-1)\,) \end{pmatrix}$;
    $ray \leftarrow ray(origin = scanhead,\ direction = down)$;
    if $M_N.Raycast(ray, \textbf{\textit{out}}\ hit)$ **then**
      $current \leftarrow hit.position$;
      if $NavMesh.SamplePosition(current, \textbf{\textit{out}}\ hit)$ **then**
        $V[i, j] \leftarrow hit.position$;
      end
    end
  end
end

Algorithm 1: Scanning the NavMesh.

$$width = \lfloor \sqrt{n * \frac{dim.x}{dim.z}} \rfloor$$
$$depth = \lfloor width * \frac{dim.z}{dim.x} \rfloor$$

3.2.2 Scanning the NavMesh

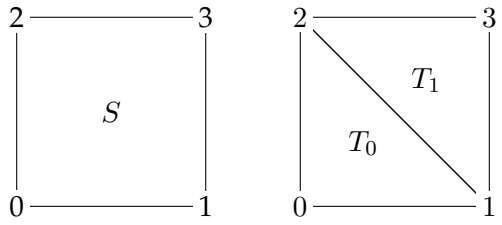
In a first step, the map positions located at the grid intersections need to be found.

The implemented method follows a 3D scanner analogy. A virtual scan head is moved across the map surface in equidistant steps and samples the surface positions of the NavMesh using raycasts.

The used algorithm can be seen in figure 1. The position of the scan head for the current iteration is calculated via linear interpolation over the minimum and maximum x and z values of the NavMeshes vertices using the normalized loop indices. Along the y-axis, the scan head is set to the maximum y value and moved up some constant value c to make sure no surface is missed by the raycast.

A ray is then projected from the scan heads position downwards. If M_N is intersected, a first estimation of the position is found. Because the NavMesh is highly simplified, the intersected position might not lie directly on the ground level of the original map. Using the NavMeshes *SamplePosition* function at the found position then projects the position onto the map.

A two-dimensional array of vertex positions is used to simplify the triangulation step discussed in the next section. In a final conversion step, the two-dimensional indices can be mapped to one dimension utilizing a dictionary.



$$v_0 = (0,0), v_1 = (1,0),$$
$$v_2 = (0,1), v_3 = (1,1)$$

Figure 3.1: Example triangulation of rectangular shape S into triangles T_0 and T_1

3.2.3 Short Introduction to Mesh Representation for Rendering

The previous step provides a grid of unconnected vertices. To render the mesh, the existing vertices need to be connected to form simple geometric shapes which can be rendered on graphics hardware.

Triangles are generally used in game engines, because they form the simplest type of polygon and always form a plane, which is not true for polygons with higher vertex count. Furthermore, most available graphics acceleration hardware is designed around triangle rasterization. [18]

The simplest way of feeding a mesh to the graphics pipeline is in the from of a triangle stream, where every triangle is defined by the three vertices it is formed by, their order defining the direction the triangle is facing. Whether clockwise (CW) or counter clockwise (CCW) order is used can generally be decided by the developer. [18] For the following example CW order will be used.

Figure 3.1 shows the simple rectangular shape S being triangulated into triangles T_0 and T_1, such that the surface normal points out of the plane. This is achieved by picking an arbitrary starting vertex per triangle and listing the vertices in CW order, giving us the triangle stream T.

$$T = (\begin{pmatrix}0\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}, \begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}, \begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}1\\0\end{pmatrix})$$

One disadvantage of this method can be seen here. Vertices shared by multiple triangles will be included multiple times. Sending duplicate data should be avoided, because GPU bandwidth is limited, and the transformation and lighting stages of the rendering pipeline would be repeated as well. Memory is also a consideration, especially given that often additional per vertex data such as vertex color or UV coordinates will be added. [18]

Indexed triangle lists are one solution to this problem, and are used for the GridMesh. Instead of defining triangles directly using the vertex data, a triangle list T of indices into the vertex array V is used.

$$V = (\begin{pmatrix}0\\0\end{pmatrix}, \begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}, \begin{pmatrix}1\\1\end{pmatrix})$$
$$T = (0,2,1,2,3,1)$$

Input: T: List of two-dimensional index vectors representing triangles

for $i \in [0, width - 2]$ **do**
  for $j \in [0, depth - 2]$ **do**
    if
     $LoS(V[i,j], V[i+1,j]) \wedge LoS(V[i,j], V[i,j+1]) \wedge LoS(V[i+1,j], V[i,j+1])$
     then
      $T.AddAll((\begin{pmatrix} i \\ j \end{pmatrix}, \begin{pmatrix} i \\ j+1 \end{pmatrix}, \begin{pmatrix} i+1 \\ j \end{pmatrix}));$
    end
    // Repeat for second triangle
  end
end

Algorithm 2: Triangulating the GridMesh.

3.2.4 Triangulating the Grid

Given that the vertices output by the scanning step form a grid, the triangulation will be similar to the simple example. However, vertices should only be connected in the resulting mesh if the positions are also connected by line-of-sight on the NavMesh, meaning that a directly line can be drawn between them without hitting a NavMesh boundary. [31]

This ensures that no triangles are formed across un-walkable map areas. The *LoS* function in the following pseudo-code implements such a function and returns true only if the line-of-sight between the provided positions is uninterrupted.

The triangulation algorithm can be seen in algorithm figure 2. At every vertex, safe for the upper most rows and columns, the two triangles forming the neighboring grid cell are checked for line-of-sight. All triangles that are connected on the NavMesh are then added to the triangle index list.

In a clean up step, useless vertices that are not part of any triangle are omitted and the indices are mapped to one dimension. The resulting vertex and triangle lists can then be combined into a mesh that can be rendered to display analysis data.

3.2.5 Extending to General Levels

In order to extend the algorithm to account for multiple levels, some adjustments need to be made.

In the scanning step, additional downward rays are spawned at the intersection forming a pillar of intersected positions. The result is a jagged two dimensional array of vector arrays, with all intersected positions in a top to bottom order. Jagged in this case describes the fact that the depth will vary across the grid according to how many layers are present at that scan head position.

In the triangulation step, vertices of different levels then need to be connected. It is assumed that a vertex is connected to at most one vertex in every neighboring pillar.

The index of the connected neighbor in such a pillar then needs to be found, if a con-

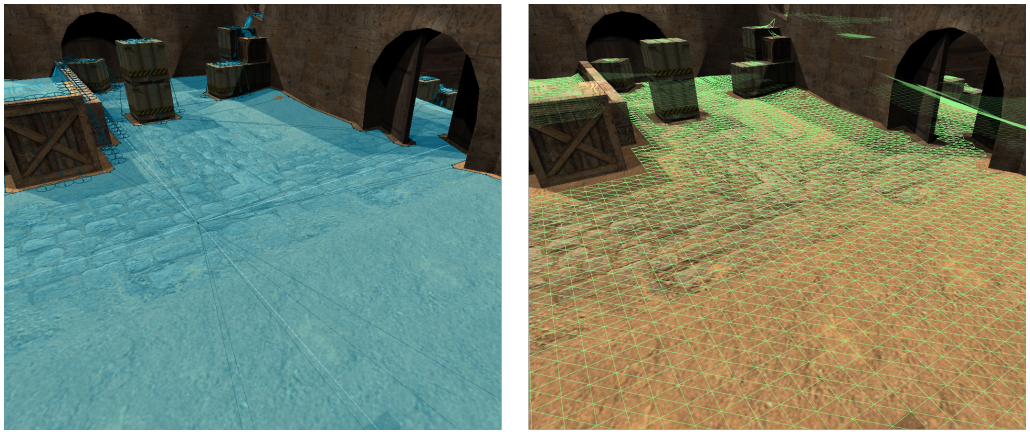


Figure 3.2: A Unity NavMesh in blue on the left and the resulting GridMesh shown in wireframe on the right.

nection exists at all. This functionality is realized in the *ConnectedVertex* function. If no connection exists an error value is returned. The triangulation algorithm can be seen in algorithm figure 3.

An additional dimension is added to the indices in triangle list T that corresponds to the index of the vertex in it's pillar. For every vertex in a pillar, the connected vertices are then evaluated. If a vertex is connected to two neighbors, and these neighbors are also connected, the resulting triangle is added.

Input:
V: Jagged two dimensional array of vertex arrays representing hits along the vertical at the scan head position.
T: List of three-dimensional index vectors representing triangles

for $i \in [0, width - 2]$ **do**
  for $j \in [0, depth - 2]$ **do**
    for $w \in [0, V[i,j].Length]$ **do**
      $u \leftarrow ConnectedVertex(V[i,j][w], V[i,j+1])$;
      $r \leftarrow ConnectedVertex(V[i,j][w], V[i+1,j])$;
      if $u \neq -1 \wedge r \neq -1 \wedge LoS(V[i,j+1][u], V[i+1,j][r])$ **then**
        $T.AddAll((\begin{pmatrix} i \\ w \\ j \end{pmatrix}, \begin{pmatrix} i \\ u \\ j+1 \end{pmatrix}, \begin{pmatrix} i+1 \\ r \\ j \end{pmatrix}))$;
      end
    end
    // Repeat for second Triangle
  end
end

Algorithm 3: Triangulating the GridMesh.

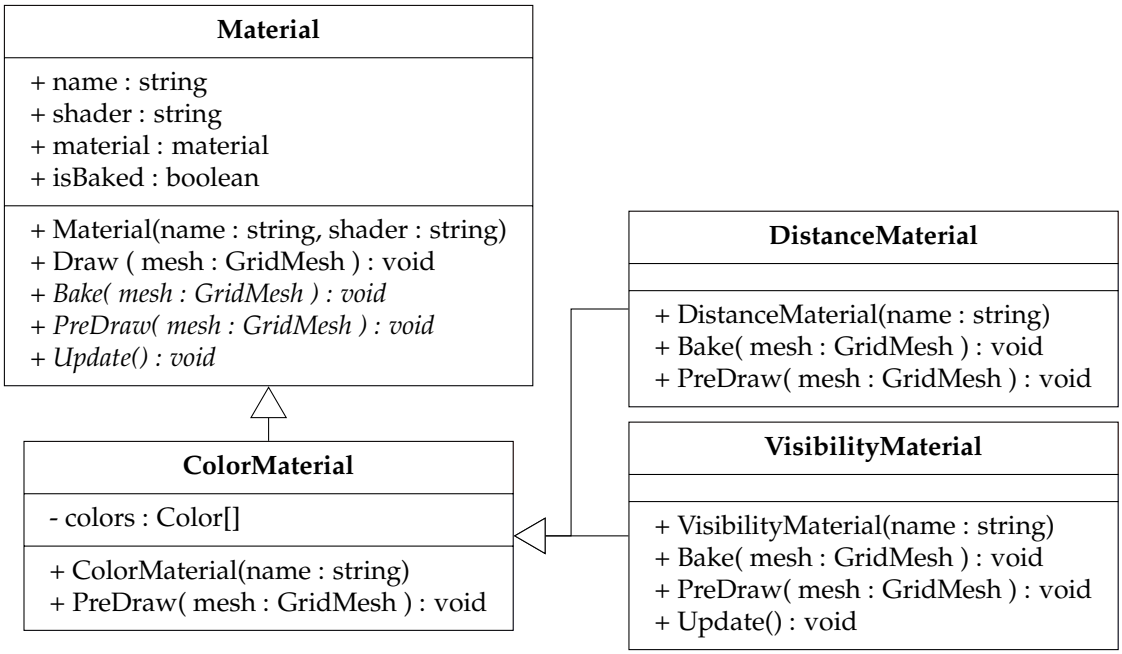


Figure 3.3: Class structure of selected GridMesh materials

3.3 GridMesh Materials and Rendering of the GridMesh

Similar to materials applied to standard Unity meshes, GridMesh materials perform the rendering of the GridMesh in the scene view.

The general structure of the materials can be seen in figure 3.3. In addition to the rendering task, GridMesh materials also implement the desired evaluation in the *Bake* function.

For rendering, the GridMesh vertex and triangle lists are combined into a Unity mesh object, which can be accessed via the GridMesh *Mesh* property. The base classes draw method then sets up the render pass and draws the mesh.

```
1 public void Draw( GridMesh mesh, float verticalOffset ) {
2 if ( isBaked ) {
3  material.SetPass(0);
4  Graphics.DrawMeshNow( mesh.Mesh, Vector3.zero + verticalOffset *
       Vector3.up, Quaternion.identity );
5  }
6 }
```

Via the shader variable, custom shaders can be supplied using their unique identifier in the Unity Editor. The Unity material used in rendering is then loaded using this identifier upon construction or deserialization.

Any additional tasks needed for rendering, such as assigning the vertex color array in the case of color materials or setting additional shader attributes, are implemented in the child classes *PreDraw* function according to the specific need and shader.

In order to prevent z-fighting issues, the mesh is rendered with a slight vertical offset which is added to the position.

An example rendering of a GridMesh material assigning random vertex colors can be

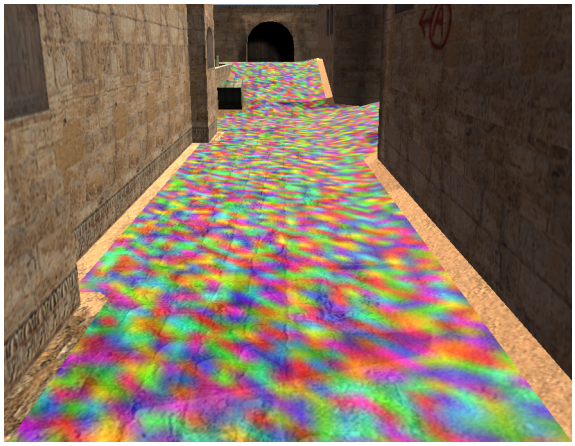


Figure 3.4: Example rendering of a GridMesh material assigning random color values to every vertex.

seen in figure 3.4.

4 Designating Map Locations

4.1 Map Areas

A map area is a subdivision of the map. On the GridMesh, areas are represented as a subset of GridMesh vertices located in the map area.

To use areas in evaluations, they need to be designated by the designer first. One way to achieve this would be a system where areas can be painted directly onto the mesh, defining the subset explicitly.

One issue with this approach is the frequent rebaking of the GridMesh. In use by a map designer, the mesh would be rebaked after every alteration to the maps geometry. During this process, new vertices will be assigned and all subsets will be invalidated. The redefinition of all areas after every alteration is inconvenient. At most the altered areas should be redefined.

Because of this, areas are defined in the form of collider meshes in this project. Whenever an area is needed by a material the GridMesh vertices located in that collider can then be evaluated.

To define this mesh, the user is able to set corner points in the scene enclosing the area and defining a polygon. Using a triangulator available for Unity, a mesh is generated from this polygon [38], which is extruded up and down to form the collider mesh.

Once an area is selected, a new corner point can be added simply by clicking the desired position in the scene view window. In case a corner has been misplaced, the last added corner, or all corners, can be cleared. One simple example area with the resulting grid mesh area can be seen in figure 4.1. The system also supports more complex area geometries with verticality as can be seen in figure 4.2.

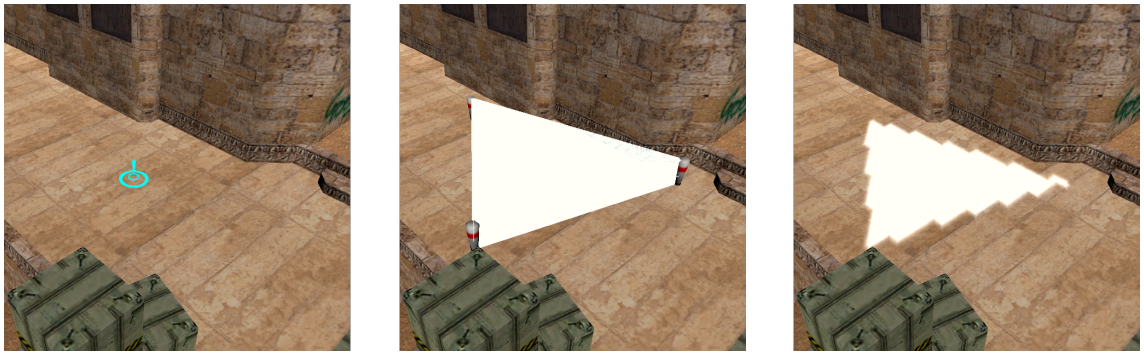


Figure 4.1: Definition of a simple area. Corners are added at the cursor location defining the collider mesh. Enclosed GridMesh vertices rendered in white in rightmost picture.

Figure 4.2: The system allows the definition of complex areas with verticality.

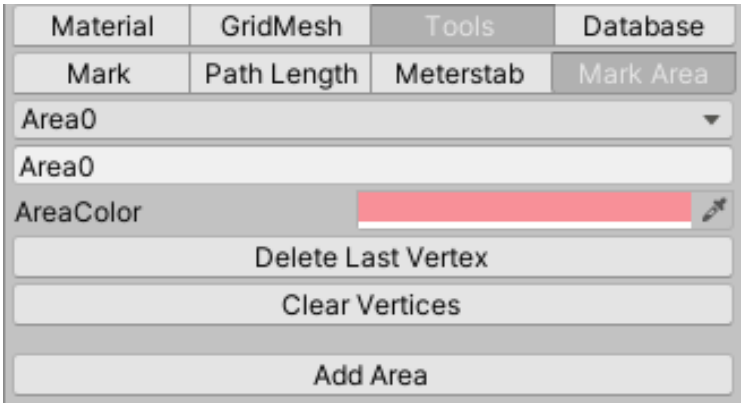


Figure 4.3: Areas can be selected, added and edited in a custom editor window

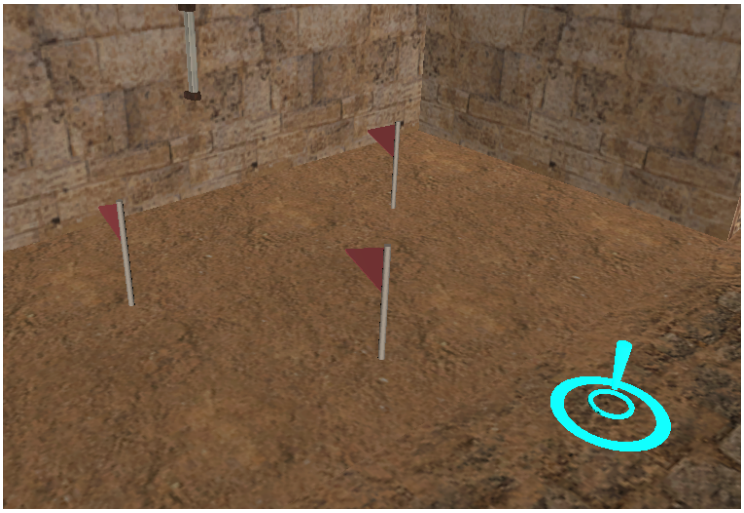


Figure 4.4: Example of a marking group added in the scene view.

4.2 Marking Groups

Marking groups are used to designate locations on the map. One example of their use is the designation of spawn locations in distance evaluations. Similar to the designation of areas, markings can be added directly in the scene view.

5 Visibility

This chapter will show how the visibility between areas can be evaluated for a source area A and target area B.

Given a position $x \in B$ on the map and source area A, the visibility of x from A is the proportion of positions in A from which a player can see another player located at x.

This question is closely related to the illumination of a point by an area light source. Treating the source area as an area light source with no intensity fall-off, traditional rendering algorithms can be used to evaluate the visibility at x.

One adjustment needs to be made to account for player geometry. While lighting algorithms usually calculate the lighting on a single surface point, the visibility of a player model is in fact an integral over the potentially visible surface of the player model.

This projects implementation is simplified in this regard. A player counts as visible, if the point located above the map surface at a defined player height is illuminated by the pseudo area light of the source area lifted by the same height.

5.1 Monte Carlo Integration

Monte Carlo integration is a method of estimating the value of an integral by randomly sampling the function in the integration interval and averaging over the results. [22]

$$F = \int_a^b f(x)dx \approx (b-a)\frac{1}{N-1}\sum_{i=0}^{N} f(X_i)$$

where:

$$X_i \in [a, b],\ N \text{ the number of samples}$$

It is extensively used in computer graphics for tasks such as global illumination [24] or area lights. [5]

5.2 Raytracing

Raytracing is one of the earliest image rendering techniques, being introduced by Appel in 1968. The method is based on the simulation of light rays falling into a virtual camera. [1]

For any point on the virtual image pane, a ray originating at the camera's origin through this point can be simulated. This ray is intersected with the scene geometry and the point p closest to the camera is returned, making a contribution to the image.

As shown by Appel, shadows cast by a point light source can then be evaluated by casting

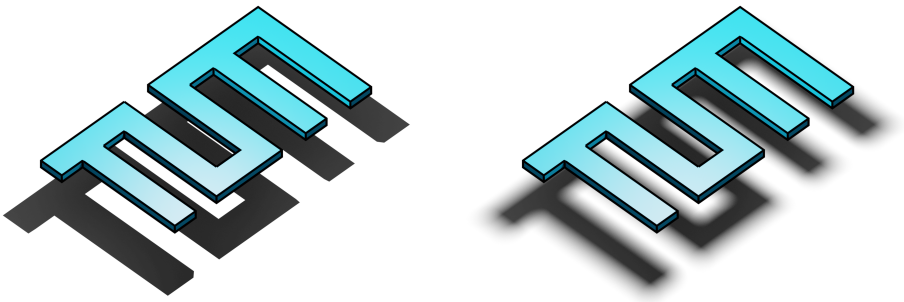


Figure 5.1: Comparison of hard shadows produced by point light sources and soft shadows with penumbra produced by area light sources (left to right). Rendered using Blender Cycles.

an additional shadow testing ray from the intersected point p to the light source. If no geometry is intersected by this ray the point is illuminated, otherwise it lies in shadow. [1]

This shadow testing is the crucial part for the task of evaluating player visibility. The target point x is predefined and independent of viewpoint, so the initial camera to scene intersection can be skipped.

Shadow testing for area light sources was introduced by Cook et al. through the use of distributed raytracing lending to the Monte Carlo approach. Instead of casting single rays to a point light source, a number of samples distributed over the area light source are taken. The proportion of un-intersected rays then yields the illumination of the point. [5] One example of soft shadows with penumbras, or partial shadows, can be seen in figure 5.1.

5.3 Area To Area Visibility

On the GridMesh, the area to area visibility for source area A and target area B is then the visibility evaluation of every target GridMesh vertex $b \in B$ from A.

Results of this evaluation are floating points values in the range $[0, 1]$, where vertices visible from every vertex in A will be assigned a visibility of 1, while fully hidden vertices are assigned a visibility of 0.

5.3.1 General Algorithm

The general algorithm used in the visibility evaluation is given in pseudo-code in algorithm figure 4.

The inputs defined by the user are the two areas A and B provided as lists of indices into the GridMesh M_G, the target number of samples N as well as the player height h.

Input:
A: source area
B: target area
M_G: GridMesh
N: number of samples
h: player height

$visiblity \leftarrow newfloat[M_G.Length]$;
$n \leftarrow min(A.Length, N)$;
for $b \in B$ **do**
  $samples \leftarrow$ randomly pick n elements from A;
  $sum \leftarrow 0$;
  for $s \in samples$ **do**
    $origin \leftarrow M_G[b] + h$;
    $target \leftarrow M_G[s] + h$;
    $direction \leftarrow (target - origin).normalized$;
    $ray \leftarrow ray(origin, direction)$;
    if $\neg Raycast(ray, ||target - origin||)$ **then**
      $sum \leftarrow sum + 1$
    end
  end
  $visibility[b] \leftarrow sum/n$;
end

Algorithm 4: General algorithm for evaluating visibility from area A to B

As the number of vertices in source area A can be below the target sample size, the actual sample size n is taken as the minimum of the number of vertices in A and N.

Then for ever vertex in B, n random vertices from A are picked. The algorithm then iterates over the sampled vertices. In each iteration the player height is added to both the source and target vertices, and the directional vector between them is calculated. With the resulting $origin$ and $direction$ vectors the ray is defined and a raycast can be performed.

To avoid counting intersections behind the target, the raycast is limited to the distance between the $target$ and $origin$ points. If no intersection is found, the sum is incremented. Once the raycasts have been performed for all sample points, the visibility is calculated as the sum divided by the number of samples.

Sample Sets

Testing this algorithm shows that the creation of unique sample sets for every target vertex is expensive. For this reason a number of sample sets are created prior to execution and used during the calculation.

5.3.2 Example

This and all subsequent examples will be based on the Counter Strike map *de_dust2* . The model used was uploaded to the 3d model sharing site sketchfab by user vrchris and is equivalent to the map featured in the original Counter: Strike from 2000. [36]

Figure 5.2: Visibility evaluation from the area marked on the left to the whole map. Visibility shown as a gradient from green, fully visible, to red, barely visible. The visibility cone from the example is marked in red in the right-most image. ($N = 804$)

de_dust2 is one of the most popular maps for the defusal game mode, currently being the fourth most played map in events covered by HLTV, an e-sports news platform centered on Counter Strike. [21].

This map is interesting to analyze because it is exceptionally fair, featuring an almost 50/50 win rate with a slight advantage for the T side. In this regard it is the fairest out of the original roster of maps, and is only rivaled by the recent addition *de_overpass* released in a 2013 update. [2]

An evaluation of the visibility from the T side spawn area is pictured in figures 5.2 and 5.3. A number of visibility cones with varying amounts of visibility can be seen, showing the visible nearby areas, as well as visible area at the bottom of the map, where the opposing teams spawn is located.

On feature worth mentioning is the slim visibility cone through a doorway marked in red which allows T side players clear a view of the connecting area between the two bomb sites. A in-game view can be seen in figure 5.4, showing the sightline which is only visible through a tight angle at specific spots. Whether this sightline was originally intended is unclear, similar spots could, however, be easily missed by a map designer.

The visibility of this area can be used by T side players to scout out where the CT side is setting up their defense and, given proper equipment, can be used engage the opposing team early on.

An evaluation of early game engagements taking place in the first 30 second of a round performed on CSGO HUB [10] can be seen in figure 5.4. The evaluation generated on CSGO HUB shows that many engagements in the early game take place in the T side spawn as well as the before mentioned area. Additionally it can be seen that T players are more likely to survive the engagements, while the results are mixed for CT players.

This shows that the evaluation does yield meaningful results in showing possible angles of engagement from a given area and can reveal tight angles. It cannot, however, predict the outcome of these engagements. Furthermore it is questionable, if pixel angles, such as presented in a previous chapter, could be found by this evaluation. These angles usually only exists in very specific spot and it is unlikely that such a spot lies on the grid.

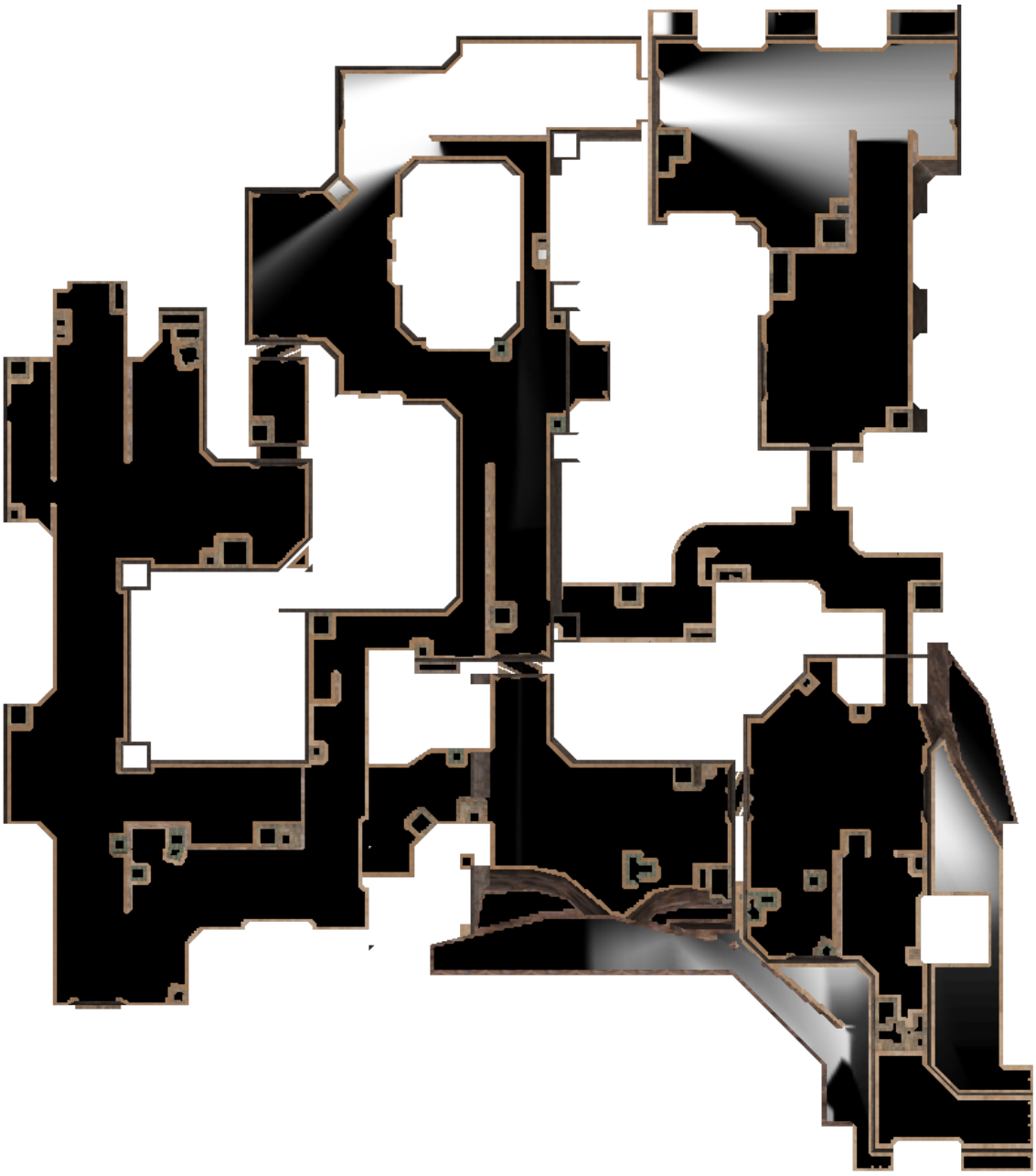
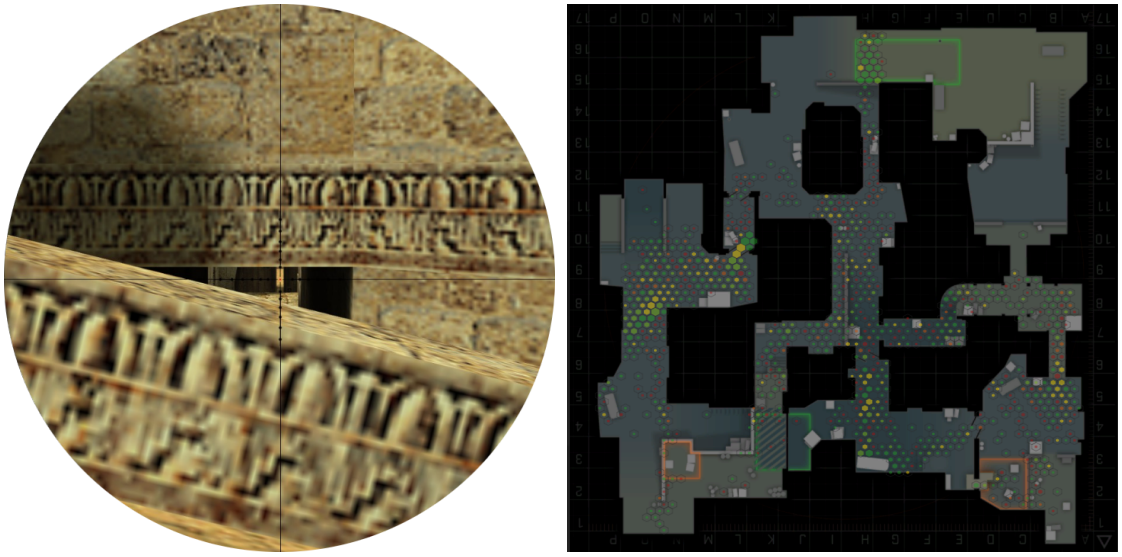


Figure 5.3: Data from figure 5.2 in grayscale for better visibility of gradients. White areas are fully visible, black areas are not visible. ($N = 804$)



(a) (b)

Figure 5.4: (a) The view from the T spawn area through the doorway in-game and (b) visualization of early engagements in the first 30 seconds of a round, showing successful engagements by T players from the spawn area (generated on GSGO HUB [10])

5.3.3 Disadvantages of the Monte Carlo Approach

While it can be shown that the estimator $\langle F^N \rangle$ tends toward the true solution F for $N \to \infty$, the error can be large if an insufficient N is chosen.

The standard deviation of the estimator is proportional to the inverse root of N, meaning that N needs to be quadrupled to half the expected error. [22]

$$\sigma(\langle F^N \rangle) \propto \frac{1}{\sqrt{N}}$$

As can be seen in figure 5.5, this can lead to extensive noise in the rendered result. Rendering the visibility at the maximal sample size 804 possible for the area gives a clear gradient of visibility values.

A rendering using a single sample set at 256 samples on the other hand shows clear scanlines, while a rendering at the same sample size using 50 sample sets appears very noisy.

This is a known issue of Monte Carlo raytracing and recent advancements made in the form of neural network driven denoising algorithms allow for a reduction of the sample size while keeping visual fidelity. [29]

The naive algorithm can, however, still be optimized to allow for overall higher sample sizes while still keeping acceptable performance.

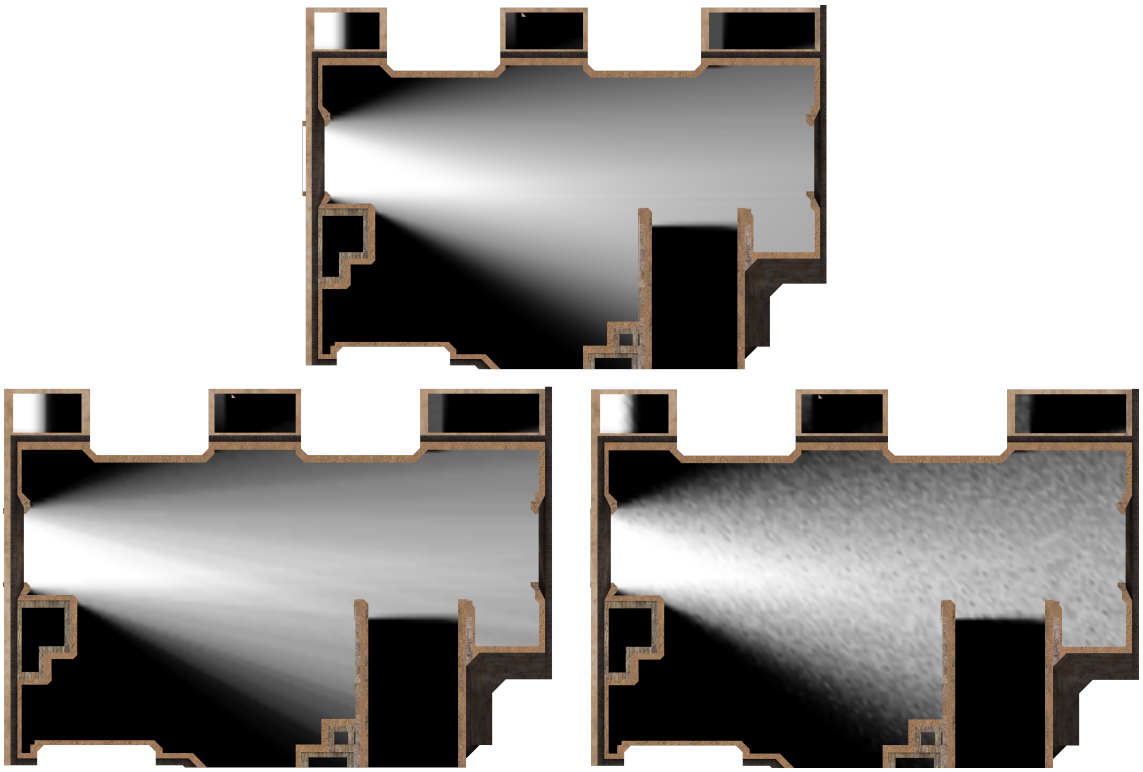


Figure 5.5: Comparison of noise at different sample sizes N. On the top $N = 804$. On the bottom $N = 256$, with one sample set on the left and 50 sample sets on the right

5.3.4 Further Optimizations

Batched Raycasts

One advantage of the Monte Carlo method is the independence of each random experiment or, in the case of raytracing, the independence of each cast ray. Therefore the raycasting can be parallelized.

Unity features *RaycastBatches* which, given a number of rays predefined in *RaycastCommands*, perform the raycasting asynchronously and parallel between frames. [13]

Due to this evaluation between frames, the algorithm cannot be implemented as linearly as seen before. Instead, the commands are dispatched in one frame and read in the next. The batched algorithm can be seen in algorithm figure 5

Here the *Schedule* function sets up the rays for all source vertices passed as a sample set. Each scheduled job then performs the raycast needed for the visibility evaluation of one target vertex. A varying number of jobs can be used to schedule the evaluation of multiple vertices per frame.

Results are made available in the *results* array when the job is completed. The sets of available and used job handles are managed through two stacks. A *vertex* stack is used to save which vertices are evaluated by the active jobs.

In addition to a performance increase, another advantage of this approach is editor responsiveness. For a small amount of jobs, the raycasting can be performed in the background with little to no decrease in responsiveness. By increasing the number of jobs the user can trade editor responsiveness for overall faster rendering times.

Adaptive Sampling

Adaptive sampling is a way of reducing the number of shadow testing rays needed to be cast into the scene, using knowledge from prior cast rays.

The concept was introduced for area light sources by Kok et al. [25] and also has applications in recent real time raytracing.[4]

As can be seen in figure 5.5, the error in the penumbra region of the visibility rendered at a low sample size is high, while the results for fully visible and fully hidden regions do not differ.

This is not a coincidence. Given a vertex with no visibility of the target area, no amount of additional cast rays will make a difference to the outcome, which will always be zero. The same is true for fully visible vertices, albeit the result will always be one.

Adaptive sampling tries to identify these vertices early and excludes them from further testing.

In practice, two sampling sizes M and N with $M < N$ are defined. Then in a first stage the visibility of all target vertices is evaluated using M samples. If a vertex is found to be fully shadowed or visible according to these M samples, the vertex will not be considered in the second stage, where for every vertex now confirmed in the penumbra $N - M$ additional rays are sent to meet the target sampling size.

While penumbra vertices can be confirmed, the method cannot confirm with absolute certainty that a vertex excluded in the first stage is not in fact in the penumbra. Therefore

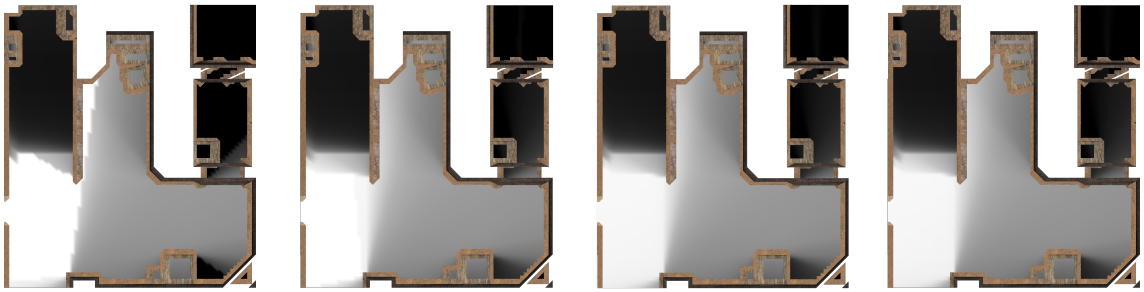
Input:
A,B, M_G,n,$visibility$: see algorithm 4,
$current$: current vertex as index into B,
$sampleSets$: array of sample sets of lenght n from A,
$jobs$: array of $RaycastBatch$ job handles,
$results$: array of results from $jobs$,
$availableJobs$: Stack of available job handles as index into $jobs$,
$usedJobs$: Stack of used job handles as index into $jobs$,
$vertexStack$: Stack of vertex index corresponding to jobs in $usedJobs$

```
// Scheduling Jobs
while availableJobs.Count ≠ 0 do
    jobIndex ← availableJobs.Pop();
    usedJobs.Push(jobIndex);
    vertexStack.Push(current);

    jobs[jobIndex] ←
      Schedule(M_G[B[current]], sampleSets[current%sampleSets.Length], jobIndex);
    current ← current + 1;
end


// Evaluating Jobs
while usedJobs.Count ≠ 0 ∧ jobs[usedJobs.Peek()].IsComplete do
    index ← usedJobs.Pop();
    sum ← 0;
    for Raycast ∈ results[index] do
        if no intersection found then
            sum ← sum + 1;
        end
    end
    visibility[B[vertexStack.Pop()]] = sum/n
end
```

Algorithm 5: Batched Algorithm for evaluating visibility from area A to B



M	Rendertime	Cast Rays	Max Absolute Error	Variance	MSE
512	67.3 s	40048640	-	-	-
128	57.6 s	28435712	0.02	0.33×10^{-7}	0.34×10^{-7}
32	41.1 s	24782144	0.12	0.13×10^{-6}	0.13×10^{-6}
8	35.3 s	19682912	0.29	0.33×10^{-3}	0.33×10^{-3}

Table 5.1: Errors and Performance of Adaptive Sampling for different M values. Errors compared to baseline $M = 512$. Images with ascending M values from left to right.

the value M should be sufficiently large to minimize errors.

A comparison of the performance as well as results of this adaptive approach can be seen in figure 5.1, compared to a baseline of $N = 2048$ and $M = 512$.

The rendering with $M = 8$ shows some problems, with penumbra regions classified as fully visible in the bottom left and as fully hidden in the bottom right. The visibility cone to a separated area on the top right is not included.

At $M = 32$, the visual error is already small, although some regions in bottom left are misclassified as fully visible. At $M = 128$, both the visual and measured error is negligible, while the speedup at this point is small but still significant.

The speedups which can be reached with this approach are also dependent on the evaluated area. If most regions of the map are hidden, the speedup will be significant. If, on the other hand, all regions of the map lie in the penumbra, no speedup will be achieved with added overhead.

6 Pathfinding

6.1 A* Pathfinding

A* is an algorithm for calculating shortest paths between nodes of a graph. It has been introduced by Hart el al. [19] in 1968, extending an earlier algorithm by Dijkstra. [12] While both Dijkstra's algorithm and A* calculate shortest paths, A* also does so optimally.

Given a graph $G = (V, E)$ with nodes V and edges E, Dijkstra's algorithm finds the shortest path between start node $s \in V$ and target node $t \in V$.

Beginning at the start node the graph is explored according to the distance to start node s. A set of *open* nodes up for exploration is kept for this purpose, with s being included initially. At every step of the iteration the node q with the minimal distance to s is removed from the set and added to a *closed* set.

Then for every neighbor p from the set of neighbors $\Gamma(q)$ of q, the distance to the start node is calculated as

$$g(p) = g(q) + d(q, p)$$

where $d(q, p)$ is the distance between q and p, usually the weight of the connecting edge. Finally if p is unexplored or the calculated distance is smaller then when it was closed, it is added to the *open* set. The process then repeats until t is found or no more nodes can be explored.

A* adds a heuristic function h, which is used to estimate the distance of the expanded node p to target node t. This function h accounts for the fact that information about the environment can be used to choose nodes which are more likely to be part of a shortest path.

Given the task of finding a shortest path from Munich to Berlin, it can be assume that such a path will not take a detour to the south. Even if Graz in Austria is closer to Munich than Leipzig, the latter will get us closer to the goal.

Then instead of choosing nodes according to calculated distance $g(p)$ the function

$$f(p) = g(p) + h(p)$$

is used.

In the real world example, the linear distance between the location and the target city can be used. Then in every iteration, the node minimizing the distance to the start and end nodes is explored.

A* is guaranteed to find the shortest path if the heuristic h is *admissible*, meaning that the heuristic never overestimates the cost of reaching the goal. [30]

$h(x) = 0$ is one example of an admissible heuristic, where A* is equivalent to Dijkstra's algorithm. The linear distance used in the real world example is another example of an

Input:
$open$: priority queue of open nodes,
$closed$: list of closed nodes,
$predecessors$: dictionary of predecessors of nodes,
s: starting node,
t: target node

```
open.Insert(s, 0);
while open is not empty do
    q ← open.DeleteMin();
    if q = t then
        closed.Insert(q);
        break;
    end
    for p ∈ Γ(q) do
        f(p) ← g(p) + h(p);
        if ¬closed.Includes(p) or f(p) less now then when closed then
            predecessors[p] ← q;
            open.Insert(p, f(p));
        end
    end
end
```

Algorithm 6: A* algorithm using a priority queue

admissible heuristic. [19]

Implementation using Priority Queues

A* and Dijkstra's algorithm are usually implemented using priority queues. A priority queue is a data structure that allows the insertion of items with a key or priority. The item with the minimum priority can then be retrieved via the $DeleteMin$ method.

One implementation of A* using such a queue can be seen in algorithm figure 6. Initially the starting node s is inserted into the queue. Nodes which are expanded are then inserted into the queue using their f value as the priority. Then calling $DeleteMin$ in the next iteration returns the node with minimal value.

The algorithm terminates either if t is found or no more nodes remain in the $open$ queue, in which case no path exists. The optimal path can than be retrieved by backtracking through the *predecessors* starting from t.

Given a implementation of the priority queue using Fibonacci heaps, Dijkstra's algorithms runs in $O(m + n\ log\ n)$ time where n is the number of nodes and m the number of edges in the graph. [15] The savings achieved by A* are then dependent on the used heuristic. [30]

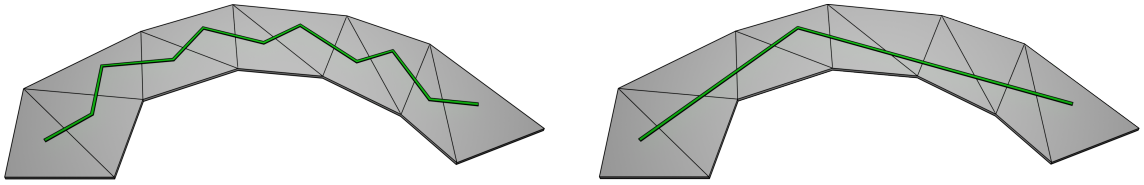


Figure 6.1: Paths are calculated between triangle midpoints and subsequently simplified. [31]

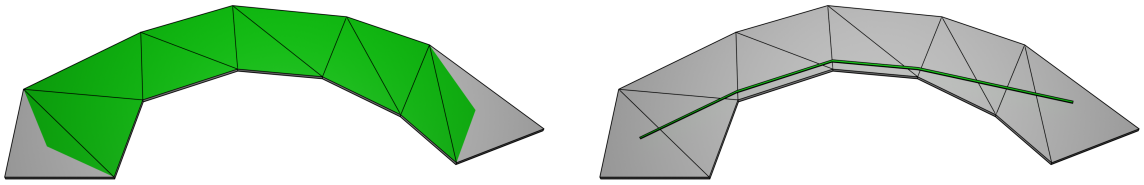


Figure 6.2: Smoothing with funneling finds the shortest path through a set of NavMesh faces. [11]

6.2 Pathfinding on the NavMesh

In this thesis all pathfinding is performed on the NavMesh structure which, in essence, provides a simplified mesh of the game world. So how can the presented algorithms for calculating shortest paths on graphs be applied to this mesh?

6.2.1 Applying A* to the NavMesh

On simple approach is presented by Snook, where the triangle midpoints of the NavMesh are treated as nodes in the graph, while graph edges exist between midpoints of triangles sharing an edge in the mesh. [31] Similarly edge midpoints or the actual NavMesh vertices can be used as nodes for pathfinding. [11]

Then, given a start and end point on the mesh, a path can be calculated. It is assumed that points lying in the same triangle can be traversed linearly, allowing paths to be evaluated between points that are not triangle midpoints.

As can be seen in figure 6.1 a path generated with this method can look unnaturally edged.

One simplification resulting in more natural looking paths is achieved by always connecting the furthest path node for which line-of-sight exists on the NavMesh, skipping intermediate path nodes. [31]

Another method is presented by Cui et al. [11]. First the path is calculated as a set of NavMesh faces connecting the start and end points, for example through the use of face midpoints as seen before. In a smoothing step a shortest path is then found for this set of faces through the use of a funneling algorithm as can be seen in figure 6.2.

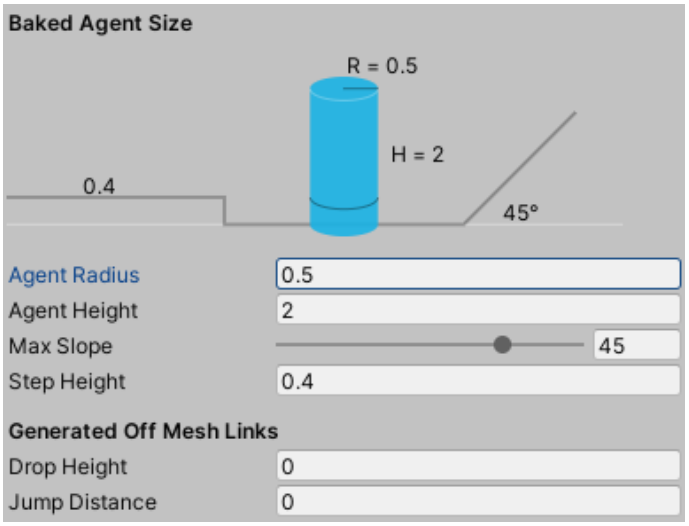


Figure 6.3: NavMesh Bake settings in Unity3D

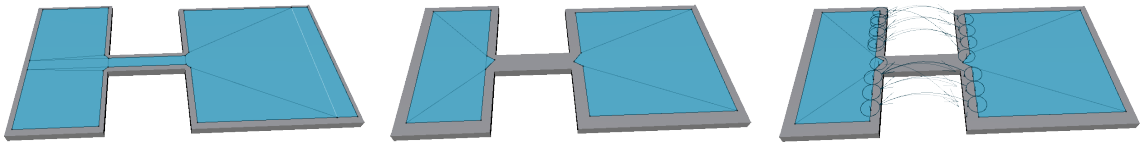


Figure 6.4: NavMeshes generated in Unity at agent width 0.125 left and 0.25 middle and right. Rightmost showcases off mesh links.

6.3 Pathfinding in Unity

NavMesh Bake Setting

In Unity a NavMesh can be baked directly in the editor for any scene. For the generation all static object present in the scene are taken into consideration. In the baking settings pictured in figure 6.3 the radius and height of the traversing agents can be set. Additionally the maximum slope angle as well as the step height that can be traversed are specified.

These settings have a direct influence on the baked mesh, which is, for example, shrunk according to the agent radius. A path that is too tight is then disconnected in the resulting mesh.

The drop height and jump distance settings are used to generate links at NavMesh boundaries. NavMesh agents can then jump across gaps and drop from heights.

The results of these settings can be seen in figure 6.4. At a small agent radius the two platforms are connected by the bridge structure, at larger radii, however, the connection no longer exists as is visible by the disconnected NavMesh in blue. With jumping distance set accordingly however off mesh links are generated connecting the platforms.

Calculating Paths in Script

Paths between two vertices can then be calculated with the *NavMesh.CalculatedPath* function. In addition to the basic pathfinding functionality, Unity also supports the definition

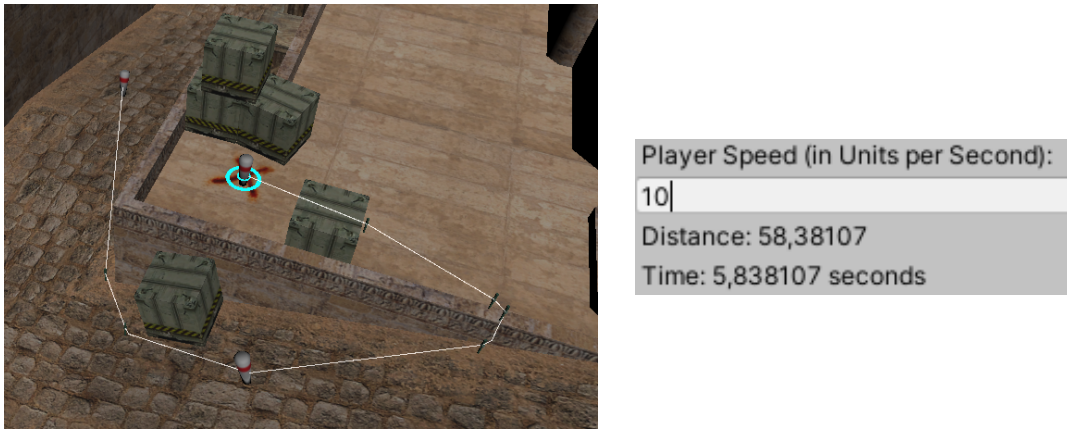


Figure 6.5: Using the single paths tool, paths can be evaluated directly in the scene view.

of areas with varying costs. Using an area mask certain areas can be excluded from the calculation.

```
1 NavMeshPath path = new NavMeshPath();
2 NavMesh.CalculatePath ( vectorA, vectorB, NavMesh.AllAreas, path);
```

The *NavMeshPath* object then includes an array of corner points of the calculated path. The path length can be calculated by summing up the distances between these corner points. In addition status information is included, such as whether only a partial path was found.

6.4 Single Path Tool

During level creation a designer might want to quickly check distances between two positions on the map. Exposing the underlying pathfinding components to the designer can therefore be valuable.

One such example would be the rotation time between bomb areas in CS. Players need to be able to get from one bomb area to the other before the bomb goes of an a 40 second timer.

The tool implemented borrows much from the earlier area definition tool. The level designer is able to place markings in the scene view. Paths are then calculated pairwise between these markings.

Results of this tool can be seen in figure 6.5. Here the red and white markings are placed by the user, while green pegs signify corners in the path. The last marking located at the cursor is updated in real time, allowing quick evaluation of multiple paths from a common starting point.

The path is then displayed directly in the scene, and the length of the path as well as the time needed for traversal given a set player speed are shown.

The placement of multiple markings allows designer to evaluate alternative paths. One example is the rotation time from bomb site B to A on *de_dust2* seen in figure 6.7.

In the example, the shortest path through the CT Spawn and CT Mid areas comes out at

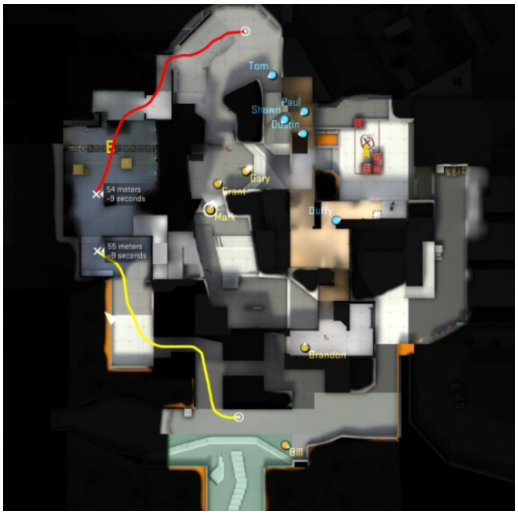


Figure 6.6: Initially a path drawing tool was shipped with Counter Strike: Global Offensive. [14]

13 seconds in rotation time. The overlayed visibility information shows, however, that a player approaching on this shortest path will be visible to enemy players located near site A for the majority of the way.

Players might therefore choose to take the alternative path through tunnels and mid, which is longer at 21 seconds, but allows players to remain undetected.

According to a forum post, most Counter Strike maps feature a shortest rotation time of 15 seconds, with *de_dust2* being closer to 16 seconds. [27]

This discrepancy in timing could be due to the fact that player speed is not in fact constant. In the specific example, a jump on the last stretch of the rotation leads to a loss of momentum that is not captured by the evaluation. The relative comparison, however, is still valuable.

Another example for the use of this tool would be a quick and dirty evaluation of meeting points from the initial spawn areas. A comparable path drawing tool included in Counter Strike: Global Offensive is recommended for this purpose in a community map design guide. [14]

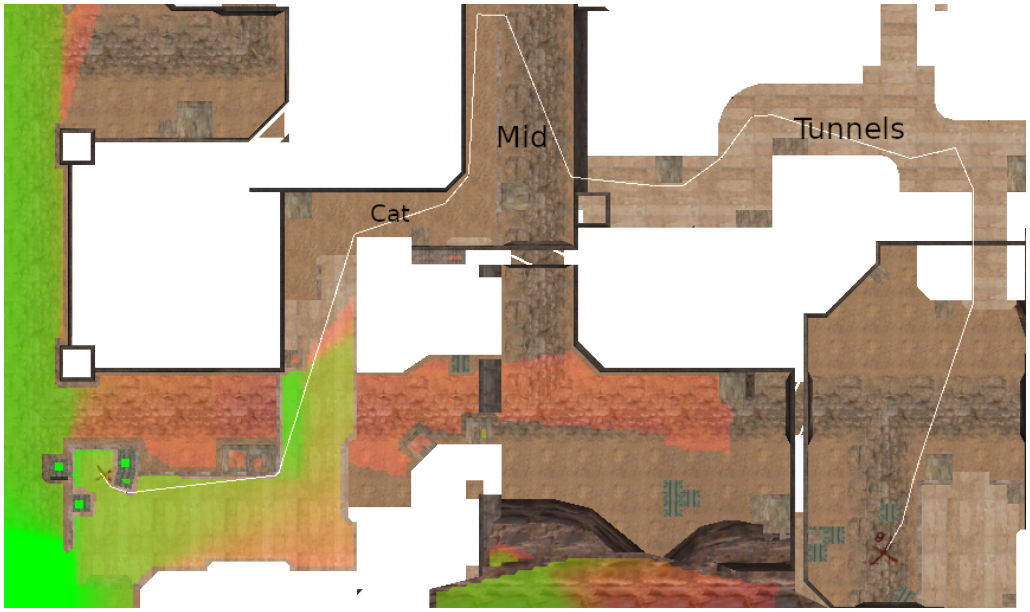
The tool included in CS:GO allows map designer to draw paths in a map view. The length of the path and the time needed for traversal are then displayed. In contrast to the tool implemented in this project, it does not feature any pathfinding component, leaving the task of estimating the shortest path to the designer. Additionally the tool seems to evaluate the length of the drawn line in the two dimensional plane without taking elevation into account.

In any case, a tool better fit for determining meeting points than any single path evaluation tool will be presented in the next section.

6.5 Two Team Distance Evaluation

The tool presented in this section, evaluates the distance to the spawn points of two teams for the entire map. The map areas reachable for the teams can then be visualized

(a) Shortest path from bombsite B to A. 13 seconds at 15 units per second.



(b) Longer path avoiding visibility by enemy. 21 seconds at 15 units per second.

Figure 6.7: Placement of intermediate markings allows designers to evaluate suboptimal paths. Visibility from bombsite A and adjacent junction overlayed.

Input:
B: set of blue team spawn positions
R: set of red team spawn positions
M_G: GridMesh

$colors \leftarrow new\, Color[M_G.Length]$;
$redDist \leftarrow new\, float[M_G.Length]$;
$blueDist \leftarrow new\, float[M_G.Length]$;

for $i \in [0, M_G.Length - 1]$ **do**
  for $v \in R$ **do**
    if $CalculatePath(v, M_G[i], path)$ **then**
      $redDist[i] \leftarrow min(redDist[i], path.Length)$;
    end
  end
  for $v \in B$ **do**
    if $CalculatePath(v, M_G[i], path)$ **then**
      $blueDist[i] \leftarrow min(blueDist[i], path.Length)$;
    end
  end
end
$maxDist \leftarrow max(max(redDist), max(blueDist))$;
for $i \in [0, colors.Length - 1]$ **do**
  $colors[i] = RGBA(redDist[i]/maxDist, 0, blueDist[i]/maxDist, 0)$;
end

Algorithm 7: Algorithm for evaluating distance from spawn for two teams

in a flooding animation, allowing map designers to see where meeting points are located and fine tune timing.

6.5.1 Calculating Distances and Visualization

Given the sets of spawn points for the two teams, here called red and blue team, the distance to spawn can be evaluated for every point on the GridMesh. The algorithm can be seen in algorithm figure 7.

The distances are calculated at every GridMesh vertex as the minimal distance to any one of the spawn points of a team evaluated using the *NavMesh.CalculatePath* function.

For rendering, these distances are then passed to the shader in the red and blue component of the vertex color. As colors in Unity are structures of RGBA color components in the range $[0, 1]$, the distance at a vertex is normalized to the maximum distance.[34] The maximum distance is then passed to the shader as an additional attribute.

In the shaders fragment function the regions with distances below a certain threshold can be rendered selectively. First the distance is reconstructed using the normalized distance values and the maximum distance.

```
1 // i : the input struct for the fragment shader stage
2 float red = i.color.r * _maxDistance;
3 float blue = i.color.b * _maxDistance;
```

A contribution is made to the fragments color if the distance is below the specified threshold and smaller than the distance to the opposing teams spawn. This allows the visualization of meeting points, with clear lines forming wherever the distances are equal. Alternatively colors can be overlapped. This allows designers to evaluate distances beyond the meeting points.

```
1 if ( red < _threshold  &&
2   i.color.r != 0.0f  &&
3   ( _Overlap || red < blue ))
4 {
5   outColor += float4(1,0,0,0.5);
6 }
```

The results of an overlapping and meeting point visualization for the example of *de_dust2* can be seen in figures 6.8 and 6.9.

In the latter, markings sourced from a community map design guide show meeting points and choke points on *de_dust2* . The meeting points evaluated by the tool closely match those outlined in the guide.

Meeting Areas

Alternatively meeting points can be visualized as areas. A point is then in the meeting area if the difference in the distances for both teams are below the specified threshold.

```
1 if ( (red - blue) < _threshold &&
2    (blue - red) < _threshold &&
3    red != 0.0f && blue != 0.0f )
4 {
5   outCol = float4 (0, 1, 0, 0.5);
6 }
```

The results of such a visualization can be seen in figure 6.10.

Figure 6.8: Visualizing distances from the spawn points for two teams with overlap. Left to right, top to bottom: 1 s, 3 s, 6 s, 8 s after spawn

Figure 6.9: Visualizing distances without overlap reveals the meeting points. Additional marks show meeting points in yellow and choke points in red according to a community created level design guide published on the steam forums. [14]

Figure 6.10: Visualizing meeting points as areas. (threshold = 25 units)

7 Processing and Combining Data

7.1 Vertex Attributes and Vertex Groups

The results generated up to this point can be put into two groups.

First, area representation on the GridMesh was discussed, which are subsets of GridMesh vertices. The second explored in the visibility and navigation chapters are per vertex attributes, which assign some floating point value to every vertex in the GridMesh.

In order to further process and combine data, a primitive database is set up, to make data globally available instead of just displaying data through the material. The data saved in the database can be seen in figure 7.1. In addition to floating point attributes and vertex groups which lie on the GridMesh, markings in the scene, which are used in the distance evaluation, are saved as off-grid groups of vertices.

Furthermore the class implements getters and setters for access, as well as GUI functions which allow the selection of attributes and groups in materials.

Any material can than simply write data to this database and access data written by other materials.

7.2 Value Range Material

The value range material is used to transform floating point vertex attributes into vertex groups. For this purpose only those vertices are selected for the group, whose value lies within the specified range.

The visibility evaluation, for example, assigns some value $[0, 1]$ to each vertex in the GridMesh. To generate a group of all visible vertices, the vertices for which this value is greater than zero are selected.

Similarly the material can be used to select all vertices within a specified distance from the spawn. Once this data is available in vertex group form, it can be reused in other materials. As areas and vertex groups are equivalent on the GridMesh, the resulting group of reachable vertices can be fed into the visibility evaluation, yielding all areas

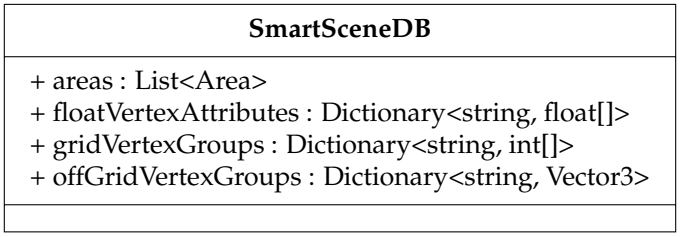


Figure 7.1: Class Structure of database

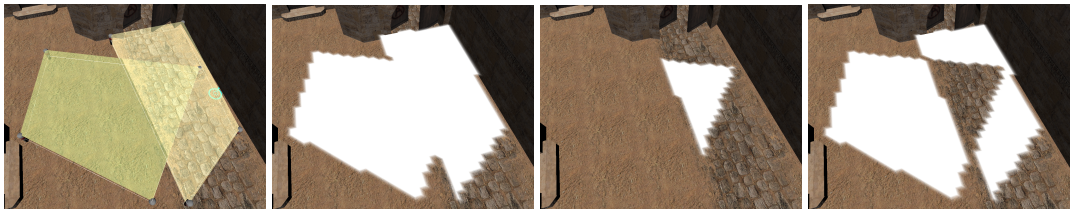


Figure 7.2: Two areas and their union, intersection and symmetric difference evaluated by the set algebra material.

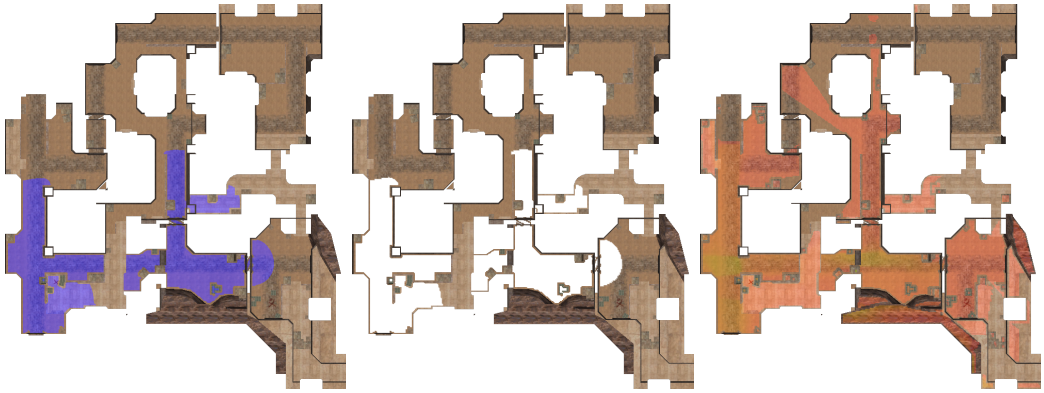


Figure 7.3: Reusing distance evaluation in the visibility evaluation using the value range material to find areas visible to CT team 7 seconds after spawn.

visible at a certain time from the round start.

7.3 Set Algebra

The set algebra material allows the combination of vertex groups using known operations of set algebra, allowing users to find the union $\cup$, intersection $\cap$ or symmetric difference $\triangle$ of any two vertex groups. An example can be seen in figure 7.2.

7.4 Example Usage

An example of the possible usages can be seen in figures 7.3 and 7.4. The goal of this evaluation is to find areas, at which T players can be attacked within seven seconds after spawn.

First the vertices reachable by the opposing CT team are evaluated using the distance and value range materials, which are then used to evaluate the areas visible to the CT team using the visibility material.

After applying the value range material to both the visibility values and the T team distance, the sets of positions visible to the CT team and positions reachable by the T team are intersected, revealing the sought-after areas.

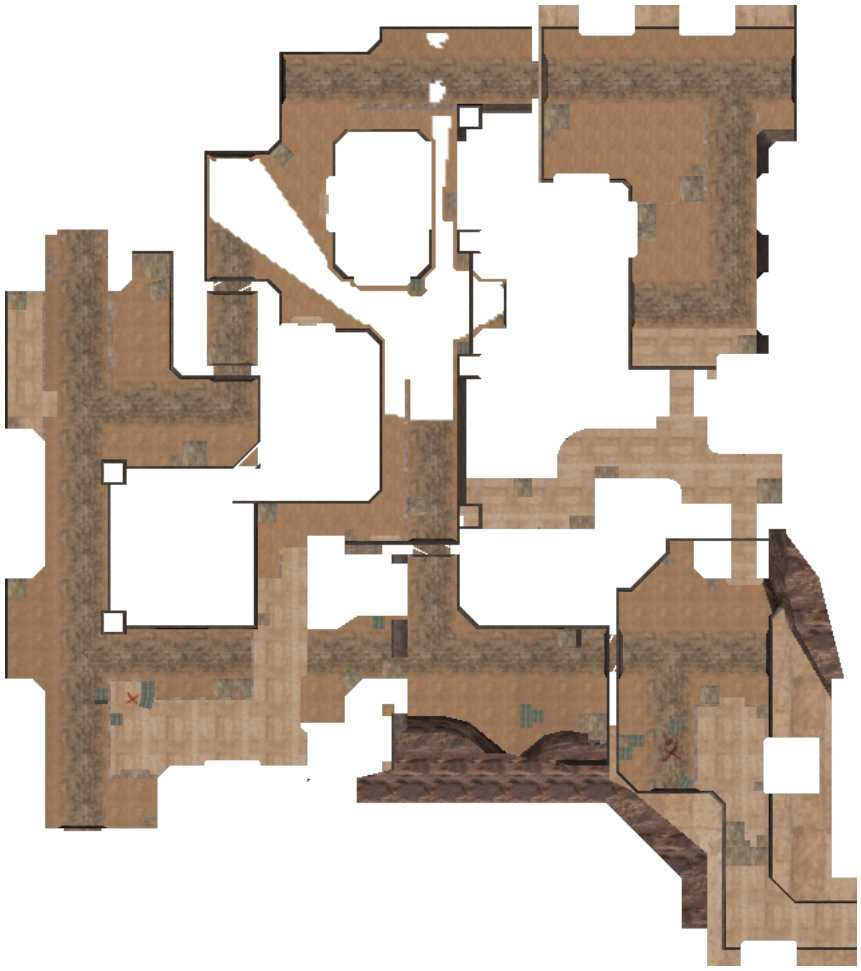


Figure 7.4: Map regions, where T players can be attacked 7 seconds into the round. 7.3

8 Related Work

8.1 Playtest Data Visualization

One area not touched upon in this thesis is the visualization and analysis of playtest data, or data collected during actual play.

8.1.1 Heatmaps

Heatmaps are commonly used to visualize different event data. Common examples are kill and death maps, visualizing the location of players at the time of these events. One example of such a classic heatmap can be seen in figure 8.2a visualizing positions of players while killing an enemy on *de_dust2* during a 2018 e-sports event. [20]

8.1.2 Aggregated Data

Paths taken by players are another example of interesting playtest data. A naive approach of visualizing paths individually will lead to visual clutter even on small data sets, as can be seen in figure 8.1.

Wallner et al. used a spacial decomposition of the game world based on player movement in order to visualize both aggregated paths as well as events.

Paths between neighboring cells can then be aggregated and visualized as lines connecting these cells, with the thickness of these lines showing the frequency at which that path is taken. The color of the underlying cell, as well as overlayed icons can be used to show additional values and event frequency. [37]

Although not utilized to visualize paths, an aggregated visualization of engagements on *de_dust2* can be seen in figure 8.2b. Here instead of a classic heatmap visualizing events in a continuous plane, the map is divided into a hexagonal grid.

Within it's grid cell, the data point is then scaled according to frequency of engagements taking place at that position while the color encodes if an engagement is more likely won

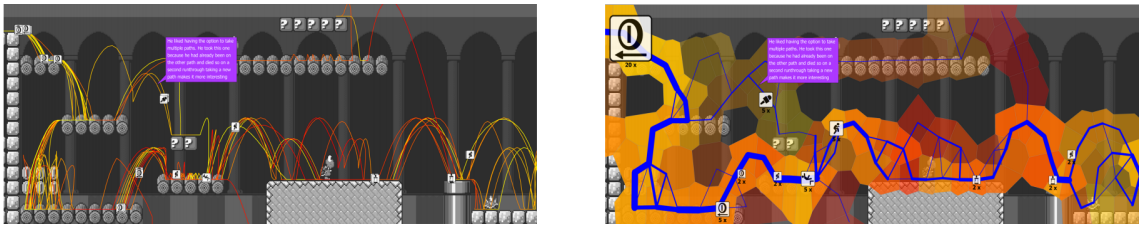
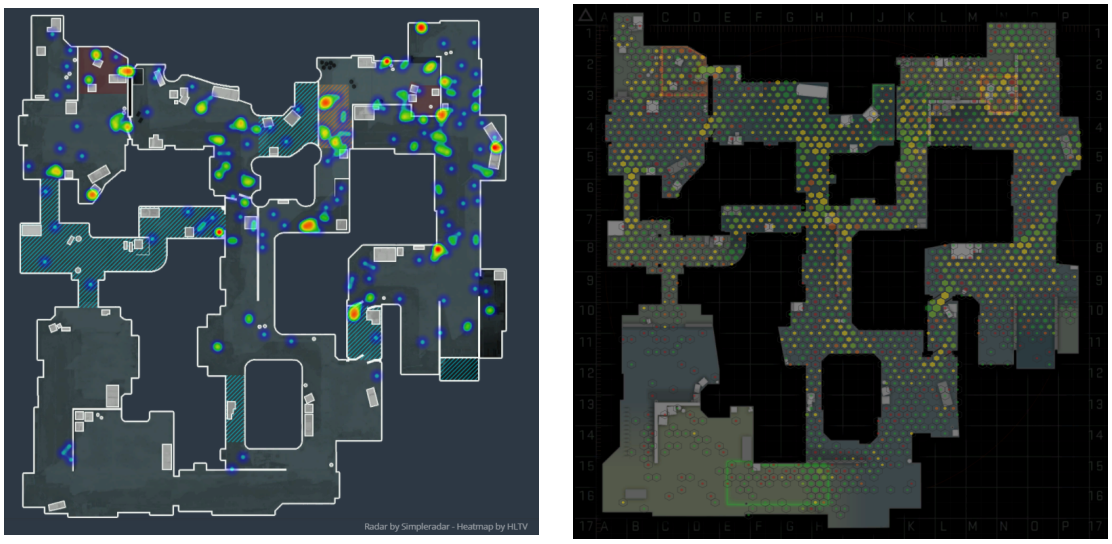


Figure 8.1: Visualization of aggregated trajectory data using a player movement based spatial decomposition by Wallner et al. [37]



(a) (b)

Figure 8.2: (a) Heatmap of killer locations during an e-sport event [20] and (b) Aggregated visualization of engagements using a hexagonal grid generated on CSGO HUB [10]

or lost. [10]

8.2 Tactical Pathfinding and Decision Making

Data structures similar to those developed in this thesis are also used in the task of tactical pathfinding. As could be seen in the navigation chapter, the shortest path might not always be the most optimal from a gameplay standpoint.

Tactical pathfinding then tries to incorporate gameplay relevant information into the pathfinding algorithms. In a 2018 paper by Makarov et al. Kill-Death (K/D) statistics as well as visibility are evaluated for cells of a Voronoi-based NavMesh and utilized during pathfinding.[26]

At each Voronoi face V_i the K/D statistic P_i can be calculated based on the amount of kills K_i and deaths D_i at this position

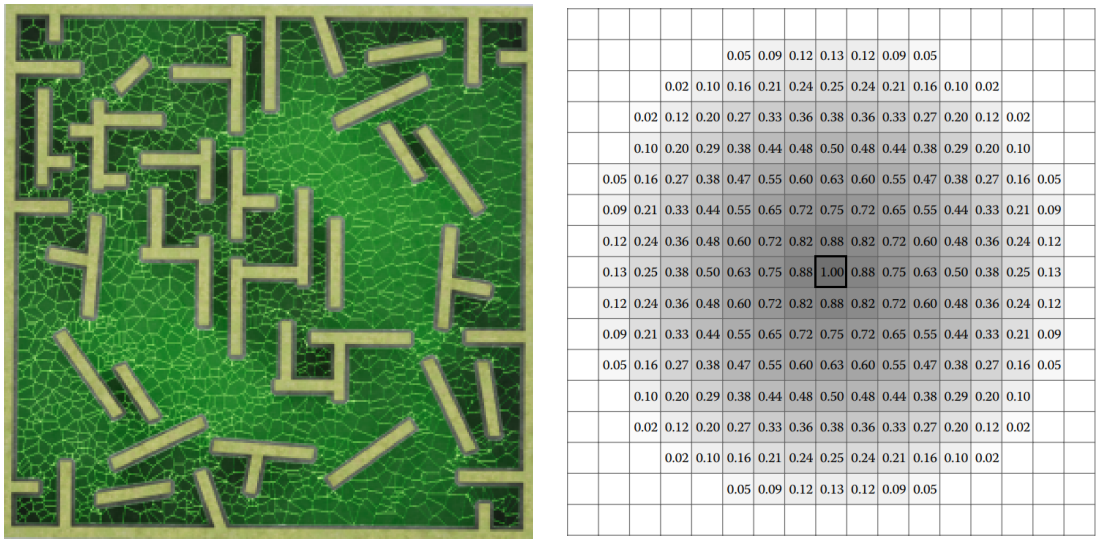
$$P_i = \frac{D_i + B}{D_i + K_i + 2B}$$

where B is the sensitivity to new events. This value then indicates the probability of being killed instead of killing an enemy while at this position.[26]

The penalty for traversing between V_i and V_j, which can be used in the A* algorithm, is given by:

$$Penalty(i,j) = V * ||S_i - S_j|| * (P_i + P_j)$$

V is the importance of that tactical property, and $S_{\{i,j\}}$ the location of the voronoi faces. The use of both K/D and visibility information has shown a 12% increase in win rate



(a) (b)

Figure 8.3: (a) aggregated visibility on NavMesh for tactical pathfinding by Makarov et al. [26] and (b) influence of an agent on a influence map [28]

compared to basic AI without additional tactical information. [26]

Opposed to the static information provided by such statistics and the analysis performed in this thesis, influence maps can be used to modify information and subsequent agent behavior at run time.

Given a grid representation of the game world, agents can assert influence over nearby cells, representing a variety of concepts such as attack distance, reachable cells in a given time frame or simply agent location.

The influences of all agents are then accumulated in the influence map. This accumulated information can be used for tactical decision making, such as choosing appropriate locations for area of effect attacks. [28]

9 Outlook and Conclusion

9.1 Possible Additions to the Tools

9.1.1 Constraints

One natural extension of the implemented tools would be constraints. This builds on the idea, that important map features such as areas and available pathways will be defined early on, similar to the sketch seen in 9.1.

Therefore, the general structure of the map is given, and the designer has already decided where spawn points, meeting point and choke points should be located, and how they are connected. Constraints can then be defined early on, to be used in automated constraint satisfaction tests during the actual building phase of level design.

These constraints can be implemented on top of the already implemented design tools.

Lets suppose an area is marked with a meeting point constraint. Then, after applying the two team distance evaluation on the map, whether an actual meeting point is located in the area can be checked.

Similarly, pairwise visibility constraint between areas can be defined during the layout phase, and checked after every iteration of the building phase.

GUI

Ideally a GUI should be composed that matches tools and practices common to level designers, and makes definition of constraints as seamless as possible.

Lets suppose map areas are represented as nodes in a graph, then a constraint could be defined between areas simply by selecting the appropriate evaluation tool and connecting the area nodes where a constraint should exists.

A level designer could then start the level design process by mirroring his design sketch in the software, defining visibility constraints and intended paths between areas. Then the only additional work needed to be done for basic constraints is the definition of areas in each iteration.

9.1.2 Playtest Data and Player Feedback

As could be seen in the related work section, playtest data can give valuable insight about how a map is played in practice.

Many issues in a maps design cannot be captured by the evaluation tools implemented. One example of this can be seen in figure 9.2, where the placement and color of windows on a scenic building made it difficult for players to make out enemies against the background, leading to a adjustment of the windows placement. [3]

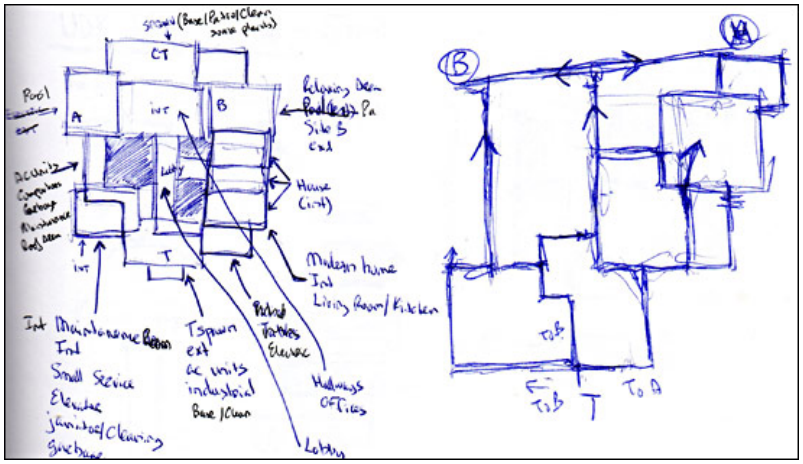


Figure 9.1: A sketch as it could be used in the early stages of designing a map, taken from World of Level Design. [17]

A system for logging and displaying playtesting data can be implemented on top of the implemented tools. As a simple example, the positions of kills and deaths could be read into the system as marking groups. A new GridMesh material can then be implemented that shows heatmaps for these events directly in the Unity Editor.

Although changes were implemented based on user feedback in the before mentioned example, these tools could be used to verify a map change has the intended effect. In this case the number of kills observed at the changed location should decrease, as an unfair advantage has been removed.

9.2 Implications for Tactical Pathfinding

As was shown by Makarov et al., the inclusion of visibility and K/D maps in pathfinding tasks can lead to good result. While a combined usage of both visibility and K/D information lead to a 12% win rate compared to a basic AI, the use of only visibility reduced that win rate increase to 3%. [26]

The visibility information available to the AI was aggregated visibility, or the visibility to all other Voronoi faces. For games which define clear spawn points and feature no respawn mechanic during the match, a temporal component can be added to the visibility function.

Lets suppose a map is fully traversable by teams *red* and *blue*, then there exists a minimal time t_f at which all map regions are discoverable by both teams. Then using the tools implemented in this thesis, the visibility $V_{\{red,blue\}}(t, P)$ of any team at time $t \in [0, t_f]$ and location P can be sampled.

For every face F_i on the NavMesh and team $T \in \{red, blue\}$ the visibility values of GridMesh vertices which lie on that face can be averaged.

$$V_{T_i}(t) = avg(\{\, V_T(t, P_j),\ P_j \in M_G \wedge P_j \text{ lies on } F_i \,\})$$

Then analogous to the penalty calculated by Makarov et al., we can define the time-dependent penalty

Figure 9.2: Adjustment of window placement to increase player contrast in Counter Strike: Global Offensive. [3]

$$Penalty_T(i,j,t) = V * ||S_i - S_j|| * (V_{T'_i}(t) + V_{T'_j}(t + \frac{||S_i - S_j||}{speed}))$$

where $speed$ is the defined player speed, and T' is the team opposing T.

It should be mentioned here, that the above penalty function is only valid for Voronoi based NavMeshes similar to the one applied in the source paper. Per definition the distance traveled in either face when traversing between source points of neighboring voronoi faces is equal. This is not necessarily true for general NavMeshes, and the visibility values would need to be weighted in accordance to the proportional distance traveled in that face.

In practice this time-dependent visibility function $V_{T_i}(t)$ can be calculated at editor time for a number of sample times $t_s \in [0, t_f]$ and interpolated during run time. For all $t > t_f$ the visibility function then degenerates to the aggregated visibility.

Whether such a time dependent penalty function would improve win rates beyond the 3% increase seen in the aggregated case would be interesting.

9.3 Conclusion

The goal set for this thesis was the creation of analysis tools, that support developers of competitive games during the game design process. With the general framework provided by the GridMesh, a common basis for such analysis tools was created, and the implemented tools showed promising results for the identification of meeting points and sightlines on Counter Strike maps.

Further research can be made in the use of generated analysis data for game artificial in-

telligence, both for tactical pathfinding and decision making tasks. Additionally, neural networks could be trained on data generated from selected maps. Whether a maps fairness, or other attributes, can be predicted by the generated analysis data would be one interesting question.

Competitive games of other genres also provide opportunities for further research, requiring analysis tools based on entirely different map design principles and gameplay systems.

Finally, whether the implemented tools would be accepted and deemed valuable by experienced map designers remains an open question.

List of Figures

List of Tables

Bibliography

[1] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 37–45, New York, NY, USA, 1968. Association for Computing Machinery.

[2] Counter-Strike: Global Offensive » Map Data. https://blog.counter-strike.net/index.php/map-data-01/, 2014. Accessed: 2020, September 8 [Archive].

[3] Counter-Strike: Global Offensive » Enemy Spotted. https://blog.counter-strike.net/index.php/2020/06/30428/, 2020. Accessed: 2020, September 12 [Archive].

[4] Jakub Boksansky, Michael Wimmer, and Jiri Bittner. *Ray Traced Shadows: Maintaining Real-Time Frame Rates*, pages 159–182. Apress, Berkeley, CA, 2019.

[5] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18(3):137–145, January 1984.

[6] Bomb Defusal | Counter-Strike Wiki | Fandom. https://counterstrike.fandom.com/wiki/Bomb_Defusal, 2020. Accessed: 2020, May 9 [Archive].

[7] Competitive | Counter-Strike Wiki | Fandom. https://counterstrike.fandom.com/wiki/Competitive, 2020. Accessed: 2020, May 9 [Archive].

[8] Counter-Strike | Counter-Strike Wiki | Fandom. https://counterstrike.fandom.com/wiki/Counter-Strike, 2020. Accessed: 2020, May 9 [Archive].

[9] Counter-Strike: Global Offensive | Counter-Strike Wiki | Fandom. https://counterstrike.fandom.com/wiki/Counter-Strike:_Global_Offensive, 2020. Accessed: 2020, May 9 [Archive].

[10] CSGO HUB. https://www.csgohub.com/match/map-analysis/detailed?map-name=de_dust2&past_months=1&round_stage=early,middle,late&rank_group=global&map_layer=0&team=2,3&economy=pistol,eco,full_buy, 2020. Accessed: 2020, September 6 [Archive].

[11] Xiao Yan Cui and Hao Shi. An overview of pathfinding in navigation mesh. 2012.

[12] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, December 1959.

[13] Unity3D Documentation. Unity - Scripting API: RaycastCommand. https://docs.unity3d.com/ScriptReference/RaycastCommand.html. Accessed: 2020, August 31 [Archive].

[14] Exodus. Steam Community :: Guide :: The dos and don'ts of CS:GO level design [Summer 2019 overhaul]. https://steamcommunity.com/sharedfiles/filedetails/?id=1110438811, 2019. Accessed: 2020, May 11 [Archive].

[15] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in

improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[16] Alex Galuzin. CS:GO 6 Principles of Choke Point Level Design. https://www.worldofleveldesign.com/categories/csgo-tutorials/csgo-principles-choke-point-level-design.php, 2013. Accessed: 2020, May 12 [Archive].

[17] Alex Galuzin. CS:GO How to Design Gameplay Map Layouts (Complete In-Depth Guide). https://www.worldofleveldesign.com/categories/csgo-tutorials/csgo-how-to-design-gameplay-map-layouts.php, 2013. Accessed: 2020, May 12 [Archive].

[18] Jason Gregory. *Game Engine Architecture, Second Edition*. A. K. Peters, Ltd., USA, 2nd edition, 2014.

[19] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[20] CS:GO Statistics database | HLTV.org. https://www.hltv.org/stats/maps/map/31/Dust2?showKills=true&showDeaths=false&sides=COUNTER_TERRORIST&sides=TERRORIST&firstKillsOnly=false&allowEmpty=true&showKillDataset=true&showDeathDataset=false&event=3885, 2018. Accessed: 2020, September 6 [Archive].

[21] HLTV. CS:GO Statistics database | HLTV.org. https://www.hltv.org/stats/maps, 2020. Accessed: 2020, September 8 [Archive].

[22] Wojciech Jarosz. *Efficient Monte Carlo Methods for Light Transport in Scattering Media*. PhD thesis, UC San Diego, September 2008.

[23] Henning Kaiser. ESL One Cologne 2018 dpa innovation. https://innovation.dpa.com/2018/08/28/keine-angst-vor-e-sport/esl-one-cologne-2018/, 2018. Accessed: 2020, September 11 [Archive].

[24] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986.

[25] Arjan J. F. Kok and Frederik W. Jansen. Adaptive sampling of area light sources in ray tracing including diffuse interreflection. *Computer Graphics Forum*, 11(3):289–298, 1992.

[26] Ilya Makarov, Pavel Polyakov, and Roman Karpichev. Voronoi-based path planning based on visibility and kill/death ratio tactical component. In *AIST (Supplement)*, pages 129–140, 2018.

[27] [CSGO] Good map rotation times - 3D - Mapcore. https://www.mapcore.org/topic/18489-csgo-good-map-rotation-times/, 2015. Accessed: 2020, September 10 [Archive].

[28] Dave Mark. Modular tactical influence maps. *Game AI Pro 360: Guide to Tactics and Strategy*, pages 103–124, 2019.

[29] Kaloian Petkov, Daphne Yu, and Helene Houle. Rendering of 4d ultrasound data with denoised monte carlo path tracing. In *ACM SIGGRAPH 2020 Posters*, SIGGRAPH '20, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice

Hall Press, USA, 3rd edition, 2009.

[31] Greg Snook. Simplified 3d movement and pathfinding using navigation meshes. In Mark DeLoura, editor, *Game Programming Gems*, pages 288–304. Charles River Media, 2000.

[32] statista. • Global eSports viewership by viewer type 2022 | Statista. https://www.statista.com/statistics/490480/global-esports-audience-size-viewer-type/, 2020. Accessed: 2020, September 13 [Archive].

[33] statista. • Sport viewership | Statista. https://www.statista.com/statistics/786734/sport-video-viewers-in-the-us-by-league/, 2020. Accessed: 2020, September 13 [Archive].

[34] Unity. Unity - Scripting API: Color. https://docs.unity3d.com/ScriptReference/Color.html. Accessed: 2020, September 12 [Archive].

[35] Unity. Wondering what Unity is? Find out who we are, where we've been and where we're going | Unity. https://unity.com/our-company. Accessed: 2020, September 13.

[36] vrchris. de_dust2 model by vrchris Sketchfab. https://sketchfab.com/3d-models/de-dust2-cs-map-056008d59eb849a29c0ab6884c0c3d87, 2017. Accessed: 2020, September 8 [Archive].

[37] Günter Wallner, Nour Halabi, and Pejman Mirza-Babaei. Aggregated visualization of playtesting data. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2019.

[38] Triangulator - Unity Community Wiki. http://wiki.unity3d.com/index.php/Triangulator. Accessed: 2020, September 11 [Archive].

[39] wikipedia. Lanxess Arena - Wikipedia. https://en.wikipedia.org/wiki/Lanxess_Arena, 2020. Accessed: 2020, September 14 [Archive].