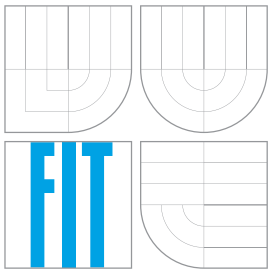
BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

REAL-TIME RENDERING OF A SCENE WITH MANY PEDESTRIANS

ZOBRAZOVÁNÍ SCÉNY S VELKÝM POČTEM CHODCŮ V REÁLNÉM ČASE

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Bc. VÁCLAV PFUDL

SUPERVISOR
VEDOUCÍ PRÁCE

HEROUT ADAM, doc. Ing., Ph.D.

BRNO 2015

Abstract

The aim of this thesis was to implement a software that would be able to render, simulate and record a scene with walking pedestrians in real-time, with emphasis on rendering level of realism. The output of the application could serve as an input test data for people counting systems or similar systems for video recognition. The problem was divided into three major subproblems: character animation, artificial intelligence for character movement and advanced rendering techniques. The character animation problem is solved by the skeletal animation of the model. To achieve the characters moving in a scene autonomously path finding(A* algorithm) and group behaviors(steering behaviors) were implemented. Realism in a scene is added by implemented methods such as normal-mapping, variance shadow-mapping, deffered rendering, skydome, lens flare effect and screen space ambient occlusion. Optimaliaztion of the rendering was implemented using octree data structure for space partitioning. Rendering stage of a scene can be easily parametrized through implemented GUI. Implemented application provides the user with easy way of setting a scene with walking pedestrians, setting its visualization and to record the result.

Abstrakt

Hlavním cílem této práce bylo implementova software, který by byl schopen vykreslovat, simulovat a natáčet scény s chodci. Výstupní videa této práce by mohla být využita jako vstupní data pro rozpoznávací video systémy. Byly implementovány metody a techniky pro vznik takovéto aplikace, jako je skeletální animace postav, vyhledávání cest(umělá inteligence), steering behaviors. Pro realistické vykrelování scén byly implementovány metody, jako je normal-mapping, variance shadow-mapping, deffered rendering, skydome, lens-flare a screen space ambient occlusion. Optimalizace vykreslujícího řeťezce byla implementována pomocí struktury octree pro dělení prostoru. Bylo implementováno GUI, které poskytuje uživateli snadné načítání a úpravu scén s chodci, nastavení vizualizace takové scény a její natáčení.

Keywords

skeletal animation, artificial intelligence, path finding, steering behaviors, real-time rendering, fast approximate anti-aliasing, lens-flare, variance shadow mapping, space partitioning, screen space ambient occlusion, gui

Klíčová slova

skeletální animace, hledání cest, umělá inteligence, steering behaviors, vykreslování v reálném čase, fast approximate anti-aliasing, lens-flare, variance shadow mapping, space partitioning, screen space ambient occlusion, gui

Cite

Václav Pfudl: Real-Time Rendering of a Scene With Many Pedestrians, master's thesis, Brno, FIT BUT, 2015

Real-Time Rendering of a Scene With Many Pedestrians

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Adama Herouta doc. Ing., Ph.D. Všechny zdroje, z kterých jsem při studiu čerpal řádně cituji s uvedením úplného odkazu na příslušný zdroj.

.....
Václav Pfudl
May 27, 2015

Poděkování

Chtěl bych poděkovat vedoucímu mé diplomove práce panu Adamu Heroutovi doc. Ing., Ph.D. za poskytnutí odborné pomoci a užitečných rad k dosažení požadovaných cílů.

Contents

Chapter 1

Introduction

The aim of this master thesis was to implement an engine which would able to simulate, render and record a video sequences of realistic scenes in real-time with walking pedestrians.

In order to achieve this goal the rendering engine had to be able to load the object models (characters, vegetation, structures, etc.), calculate the paths for the characters so they avoid static obstacles(path finding) and dynamic obstacles(steering behaviors), animate the characters (skeletal animation), render the scene using rendering technigues such as screen space ambient occlusion, normal-mapping, variance shadow-mapping, lens-flare, skydome, fast approximate anti-aliasing, etc(section 4) to achieve realistic look of the scene in real-time. Optimalizations had to be done so the calculations could run in real-time, therefore the engine had to provide some kind of visible determination(view frustrum culling). The functionality was embedded into graphical user interface to provide a user with easy way to load and customize scenes, create the movement of the characters(setting paths, adjusting steering behaviors), adjust the visualizations of the scene(customizing shader effects) and record the scenes.

Chapter 2

Methods and Technigues for Character Animation

When simulating walking pedestrians the models of such pedestrians(characters) are needed along with information about their animations(for example idle state, walking state, etc.).

2.1 Character models

Model of a character is composed by vertices (points in 3D space), each vertex includes information about its surface normal, texture coordinates, position in 3D space, tangent and bitangent(for normal-mapping purposes 4.2). These vertices can be grouped to form a single object commonly called a mesh. Model of a character then can be one solid mesh or a combination of those meshes 2.1.

There are several pieces of software designed to create character models. In this thesis Blender[1] was used for modeling and animating characters. Blender is crossplatform free to use 3D modelling software. It supports the entirety of the 3D pipeline modeling, rigging, animation, simulation, rendering, compositing and motion tracking, even video editing and game creation.

[1] for more info or tutorials visit: http://www.blender.org/

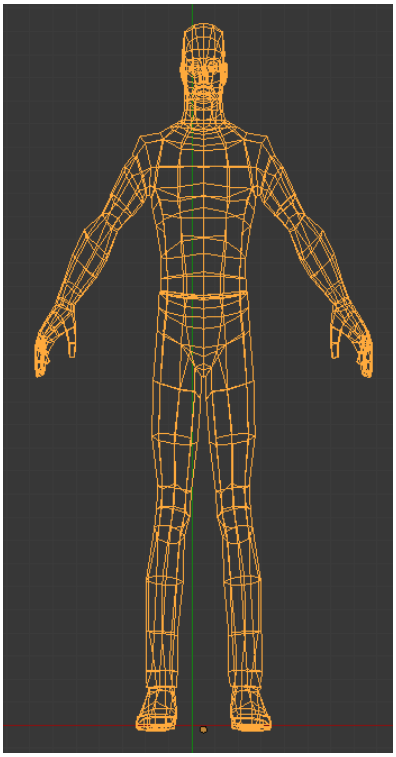


Figure 2.1: An example of character modelled in Blender by defining a set of points in 3D space (vertices).

2.2 Skeletal animation

Once a character model is available we can start animating this model. To animate it we need to define a skeleton of this character. The process of making the skeleton is called Rigging. During this process the skeleton of bones underneath the mesh is created. The mesh represents the skin of the character and the bones are used to move the mesh in a way that would mimic actual movement in the real world 2.2.

This movement is possible once we assign each vertex of the mesh to one or more bones. We then define a weight which determines the amount of influence, that bone has, on the vertex when moving. The sum of weights of the bones for one vertex should be one. That means if vertex is influenced by two bones in the same way, the weights should be 0.5 for both of these.

We can now animate our character by positioning weighted bones of created skeleton. We create set of key frames which contain the transformations of all bones in different time. When rendering a character animation, linear interpolation 2.2 is used to calculate needed transformation of a bone in time between frames.

$$x \cdot (1 - t) + y \cdot t \tag{2.1}$$

- x is the position of a bone in previous frame
- y is the position of a bone in next frame
- t is interpolating factor

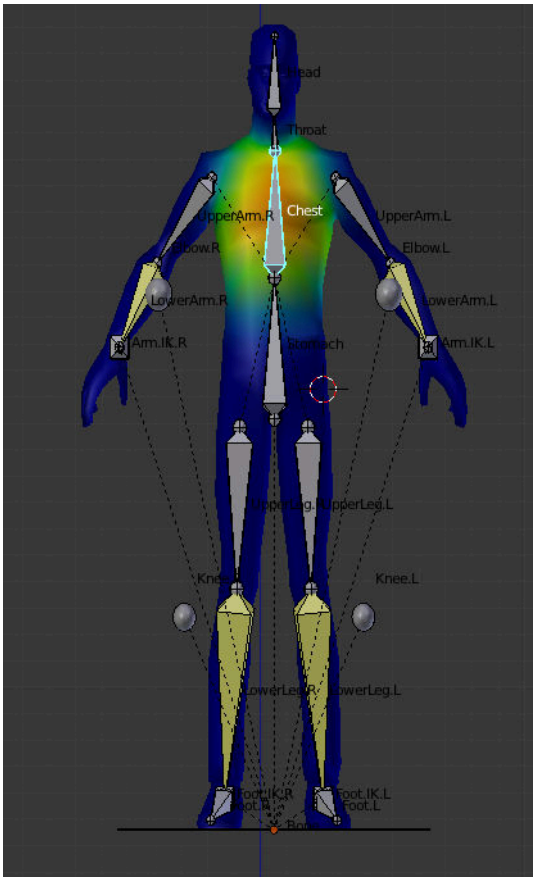


Figure 2.2: This figure demonstrates weighted skeleton created in Blender, particulary how vertices around chest are influenced by the „chest“ bone.

To calculate a bone transformation in a certain time, we need to know a frequency of the frames in current animation so we can interpolate between them.

Inverse Kinematics

Key framing transformations of the bones is usually performed by inverse kinematics. Skeleton is most often hierarchical, when we move a bone that has children, we have to move these too which is inconvenient for bigger structures. Therefore we use inverse kinematics. This allows us to operate with so-called IK bones. These bones are not part of the skeleton, they serve only as the targets skeleton bones turn to. The figure 2.3 demonstrates how the rotations of skeleton bones' joints are calculated.

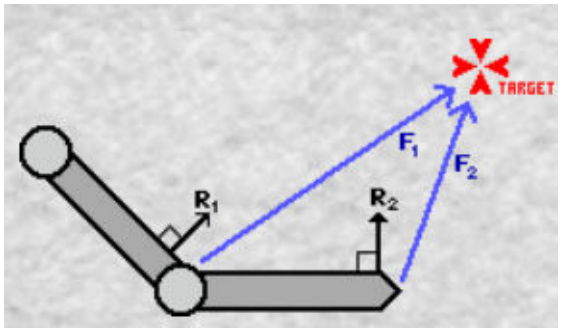


Figure 2.3: For simplicity this picture illustrates 2D inverse kinematic, where we add the dot product of force vector F and right vector R to joint of the current bone, we repeat this for every bone bound to IK bone.

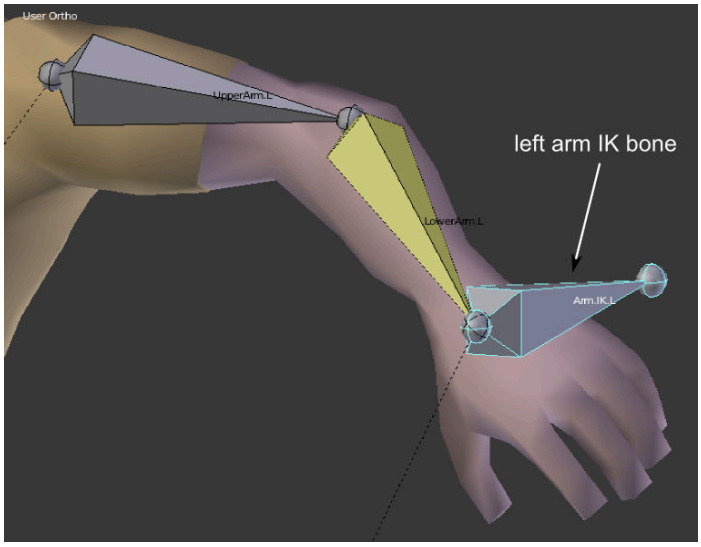


Figure 2.4: Example of left arm bones moving towards IK bone of left arm.

2.3 Format for Mesh Loading

After creating models of the scene and the characters with their animations, we need to export and load our models, animations, etc. to implemented engine. For this purpose it is necessary to choose viable file format which enables to describe all needed information (models, animations, materials etc.). Even if chosen file format supports all needed features, it might be a little problematic to parse this format to load information we need for rendering. Formats capable of doing what we need are ussually very complex and writing own parser would be out of the scope of this thesis. Therefore a library is needed to do so. In this thesis ASSIMP[2] library was used for model loading.

File format COLLADA

File format COLLADA satysfies our needs in form of export from Blender and easily accessible free to use import libraries(ASSIMP). COLLADA [3] defines an XML-based schema to

[2] http://assimp.sourceforge.net/
[3] for more information read: https://www.khronos.org/files/collada_spec_1_4.pdf

make it easy to transport 3D assets between applications - enabling diverse 3D authoring and content processing tools to be combined into a production pipeline. The intermediate language provides comprehensive encoding of visual scenes including: geometry, shaders and effects, physics, animation, kinematics, and even multiple version representations of the same asset [3].

Chapter 3

Artificial intelligence for Character Movement

Artificial intelligence (AI) is essential for moving characters from their start position to their goal position. The problem that algorithms for AI have to solve is to find a shortest path between start and goal positions and avoid all obstacles along the way.

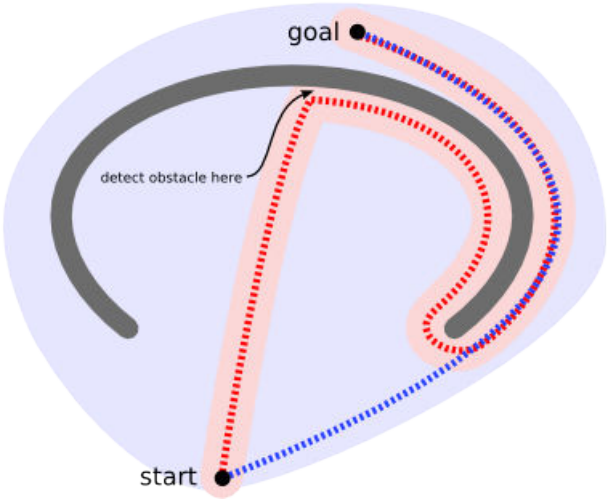


Figure 3.1: Illustration of path finding problem where character has to move around the obstacle between the start and the goal.

Figure 3.1 demonstrates how movement algorithm would work:

- in each step we move a unit one step closer to the goal
- when algorithm detects obstacle it finds a way around it

Blue line indicates the result of path-finding algorithm. Although movement algorithms are faster than path-finding ones, they can get stuck. Path-finding algorithms let us plan ahead rather than waiting until the last moment to discover there is a problem [13].

3.1 Path-Finding

All path-finding algorithms work on graphs. The graph in mathematical sense is pair $G = (V, E)$ where V is set of vertices (nodes) and E is set of the edges (links) connecting the vertices. Most of commonly used algorithms are heavily based on Dijkstra's path-finding algorithm 3.1.

Dijkstra's Algorithm

This algorithm works by visiting nodes in the graph starting with the object's starting point. It then repeatedly examines the closest not-yet-visited node, adding its neighbouring nodes to the set of nodes to be examined. It expands outwards from the starting point until it reaches the goal. Dijkstra's algorithm is guaranteed to find a shortest path from the starting point to the goal, as long as none of the edges have a negative cost (price for moving from current node to the next).

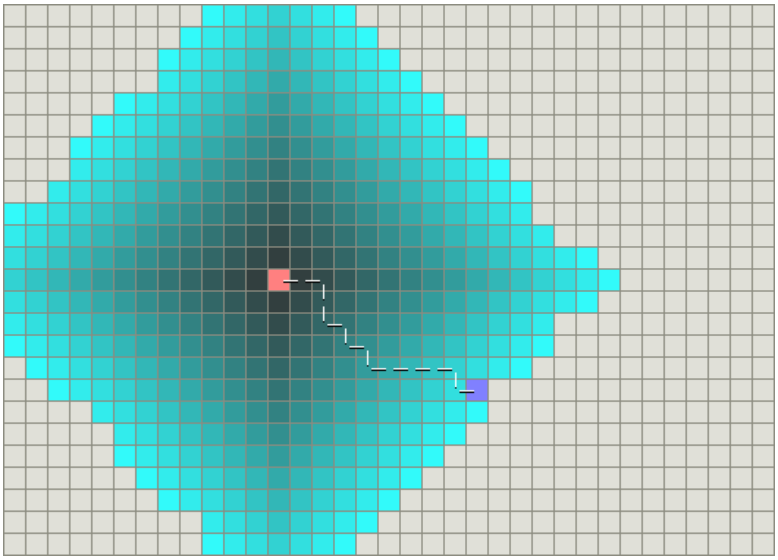


Figure 3.2: Figure demonstrates what state space was examined before the goal (blue dot) was found.

A* Algorithm

A* star algorithm is able to find shortest path just like Dijkstra's does. Though it works differently. It's functionality is based on finding a closest node with the lowest value f:

$$f = g(n) + h(n) \tag{3.1}$$

In equation 3.1 we calculate cost of neighbouring node n by summing up function values:

- $g(n)$ defines cost from start to current node + cost from current node to one of neighbouring nodes n
- $h(n)$ is commonly-named heuristic function which defines cost from neighbouring node n to our goal node

Heuristics for A* algorithm

Choosing good heuristic function is very important for A* algorithm, $h(n)$ being allways 0 degrades A* to Dijkstra's algorithm. If $h(n)$ is greater that cost of traveling from node n to goal A* is not guaranteed to find a shortest path. Other extreme is when $h(n)$ is too high relative to $g(n)$ then A* becomes Greedy Best-First-Search(we calculate only with $h(n)$).

What type of heuristic function to use depends on how the unit moves on designated map, for example if we can move only in four directions from the current node we use so-called **Manhattan distance**:

$$dx = |node.x - goal.x| \tag{3.2}$$
$$dy = |node.y - goal.y| \tag{3.3}$$
$$h(n) = dx + dy \tag{3.4}$$

If unit can move also in diagonals we use **Diagonal distance**:

$$dx = |node.x - goal.x| \tag{3.5}$$
$$dy = |node.y - goal.y| \tag{3.6}$$
$$h(n) = max(dx, dy) \tag{3.7}$$

For unit moving in Euclidean space we use Euclidean distance etc.

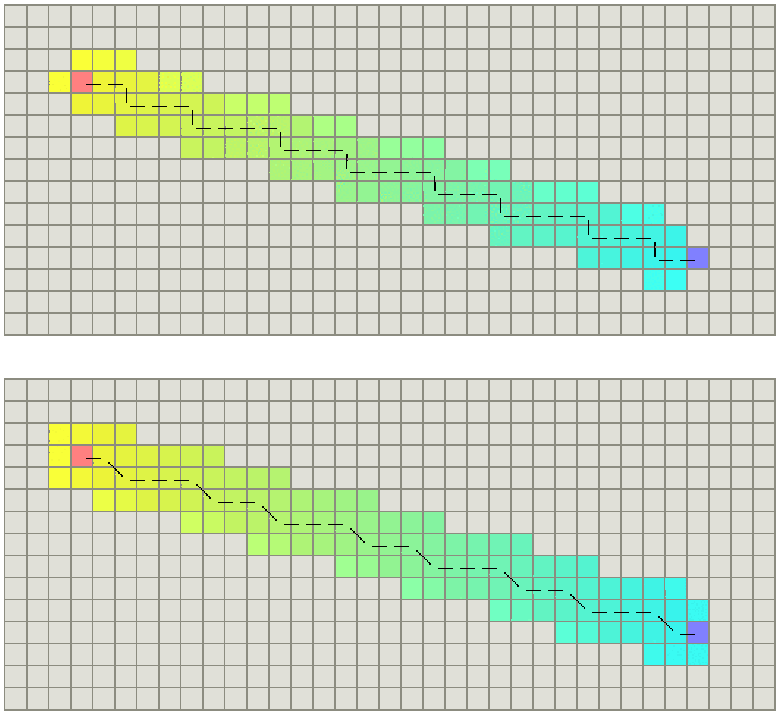


Figure 3.3: The figure illustrates path found by A* using Manhattan distance for heuristic(top) and using Diagonal distance(bottom).

3.2 Map representations

When designing path finding AI for character movement choosing map representation [13] of the world, the characters move in, is very important. It severely influences performance and quality of the found path. Path-finding algrotihms work on graphs (we search path by going through graph nodes) 3. Therefore chosen representation of the nodes affects how characters move in scene. There are several ways how we represent nodes in map representations depending on their structure.

Grid map

In grid map 3.6 path-finding algorithm searches through cells(representing nodes of search space). Algorithm is able to search in maximum of eight directions. Grid map in it's basic form is not suitable for realistic movement of characters. Moving through it can lead to „zig-zag" movement which is undesired effect when simulating realistic movement in Euclidean space. On the other hand the pros of using grid maps for path-finding are the ease of implementation and it's efficiency considering other approaches like navigation mesh 3.2.

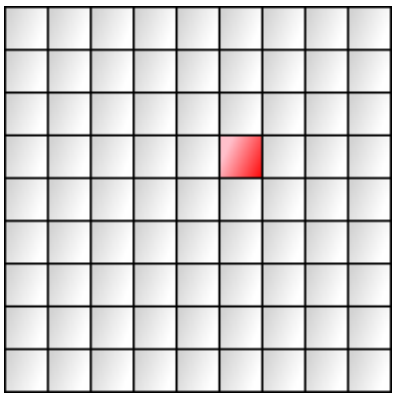


Figure 3.4: Each cell in grid map represents one node in search space.

Map capability

Map capability allows to determine which areas of the map are traversable by an agent (character) of some arbitrary size. This can be achieved by assigning a clearance value to each node of the map. A clearance value is a distance-to-obstacle metric which is concerned with the amount of traversable space at a discrete point in the environment(grid map representation). The process of measuring true clearance for a given map tile is straightforward: Surround each tile with a clearance square (bounding box really) of size 1×1. If the tile is traversable, assign it an inital clearance of 1. Next, expand the square symmetrically down and to the right, incrementing the clearance value each time, until no further expansions are possible. An expansion is successful only if all tiles within the square are traversable. If the clearance square intersects with an obstacle or with the edge of the map the expansion fails and the algorithm selects another tile to process. The algorithm terminates when all tiles on the map have been considered. Figure 3.5 demonstrates the process and the result of clearance computations.

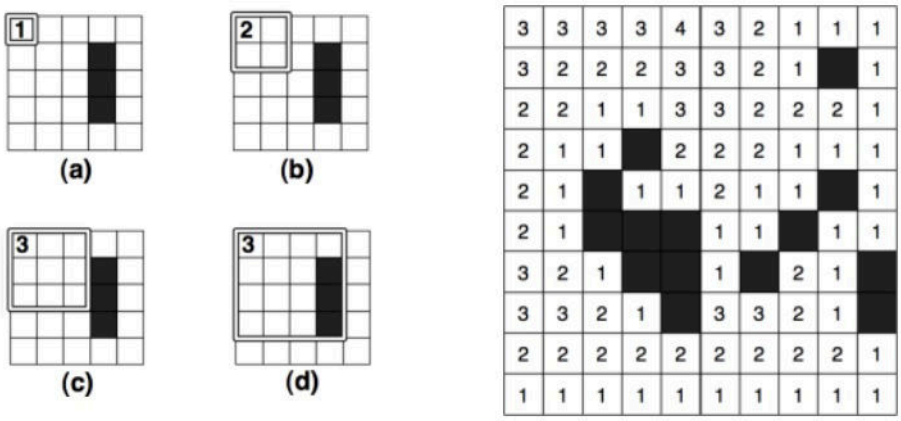


Figure 3.5: Calculation and calculated map clearance.

Navigation mesh

Navigation mesh [20] allows characters move much smoother without the result of "zig-zag" movement. It is composed by two-dimensional convex polygons which define areas of a map traversable by characters. Each polygon represents a single node which is linked to other nodes that are adjacent to it. It eliminates unnatural movement patterns thanks to convex nature of areas, where a character can move in the straight line from the point A to the point B in terms of a single area. Apart from smoother movement as a result of the path-finding using navigation mesh, it also saves search space, where large area without obstacles can be represented by a single polygon, and therefore there is no need to find a path(the result is a straight line between two points). Figure 3.6 demonstrates constructed navigation mesh. Even though we get more realistic results using this approach, there are still cons of using this method, especially problematic construction(concave polygons, unconnected nodes, etc.) and it's inefficiency compared to grid based maps.

Figure 3.6: Navigation mesh.

3.3 Characters as Autonomous Agents

When a proper map representation of the environment is chosen, we can ensure that path-finding algorithm will take care of static obstacle avoidance. To simulate characters moving in scenes there has to be a way for characters to percieve their environment and make autonomous actions, how to act in it. The term autonomous agent(character) [17] generally refers to an entity that makes its own choices about how to act in its environment without any influence from a leader or global plan. There are three key components of autonomous agents:

- an autonomous agent has a limited ability to percieve environment
- an autonomous agent processes the information from its environment and calculates an action
- an autonomous agent has no leader

In the late 1980s, computer scientist Craig Reynolds developed algorithmic steering behaviors for animated characters. These behaviors allowed individual elements to navigate their digital environments in a "lifelike" manner with strategies for fleeing, wandering, arriving, pursuing, evading, etc. Used in the case of a single autonomous agent, these behaviors are fairly simple to understand and implement. In addition, by building a system of multiple characters that steer themselves according to simple, locally based rules, surprising levels of complexity emerge. The most famous example is Reynolds's "boids" model for "flocking/swarming" behavior.

Steering Behaviors

The motion of idealized agent can be described as a series of three layers:

- action selection
- steering
- locomotion

Action selection referers to selecting an action or combination of actions based on goal or goals. Reynolds' paper describes many goal and asociated actions such as: seek a target, evade an object(other agent), follow path, etc.

Steering Once action is selected, the agent has to calculate its next move. This next move is represented as a force, more specifically, steering force:

$$steeringforce = desiredvelocity - currentvelocity \tag{3.8}$$

Figure 3.7 demonstrates an agent with current velocity and target(desired) velocity.

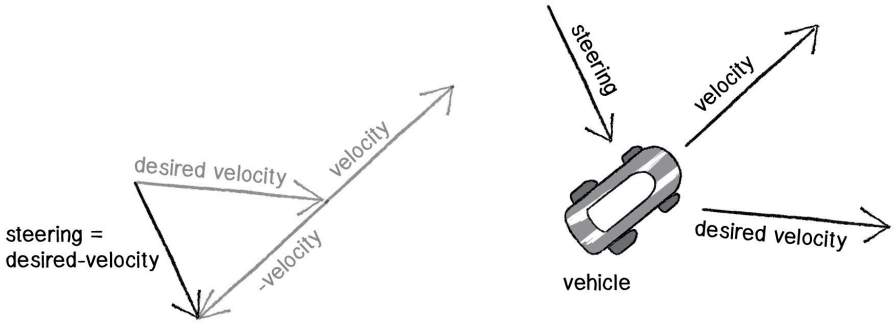


Figure 3.7: Calculation of steering force(seek a target) for an agent(vehicle, character, etc.)

Locomotion describes how an agent moves, in the case of a character, the locomotion can be described with exactly one vector(desired steering force). If a vehicle model would be considered, there would be set of vectors describing the movement(steering wheel, accelerator, brakes, etc.).

Steering behaviors are force vectors and therefore it is possible to add more than one force together to calculate desired steering simply by the sum of all participating vectors(vectors used to achieve a goal). The movement of an agent, who wants to move towards its target and also keep the distance from other agents, using steering behaviors can be calculated as follows:

$$steering = seek(target) \tag{3.9}$$
$$steering = steering + separate(agents) \tag{3.10}$$
$$velocity = velocity + steering \tag{3.11}$$
$$position = position + velocity \tag{3.12}$$
$$\tag{3.13}$$

- steering is 2D or 3D vector describing desired velocity
- seek() is function that returns steering vector towards target(as in figure 3.7
- separate() is function that returns separation vector from all other agents or agents in certain radius from an agent, as described in section 3.3 and demonstrated in figure 3.8
- velocity is 2D or 3D vector, an agent moves in direction of this vector
- position is 2D or 3D vector describing the current position of an agent

Separation Behavior

Separation behavior is the first introduced behavior(group behavior) which demonstrates an agent percieving the surrounding environment and acting according to the current state of that environment. It is based on seek behavior demonstrated in figure 3.7 but it takes the negative vector to target(flee behavior). When considering all surrounding agents as targets, from the average of all calculated flee vectors we get separation behavior.

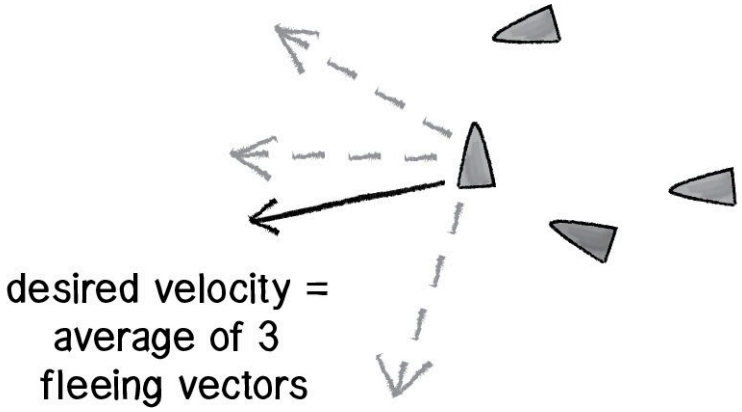


Figure 3.8: An agent acting(moving) in reaction to its environment, in this case trying to keep distance from other agents in group or in given distance around him.

In similar way to separation behavior 3.3 other behaviors can be calculated, combining such forces can lead to complex movement patterns.

Chapter 4

Realistic Real-Time Rendering Technigues

4.1 Shadow-mapping

Shadow-mapping [16] is a method introduced by Lance Williams in 1978 by which shadows are added into 3D scene. This method is suitable for real-time applications and dynamic scenes. Figure 4.1 demonstrates the process of casting shadows by objects hit by light rays.

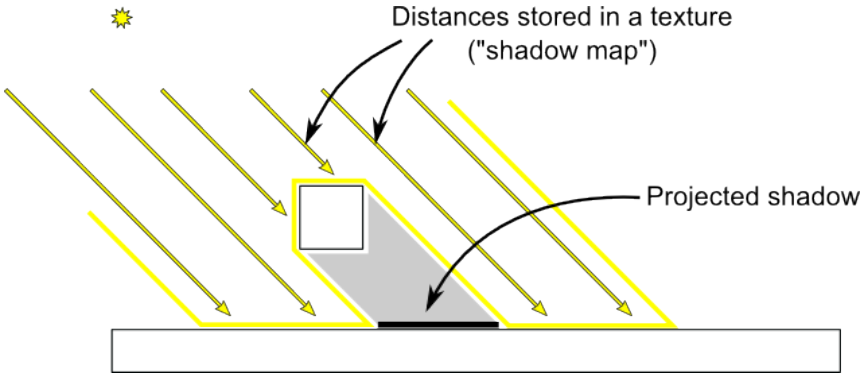


Figure 4.1: The process of shadow-mapping.

This method works in two passes. In first, the scene is rendered from the point of view of the light. Only the depth of each fragment is computed and saved into shadow map(using render to texture technigue). Next, the scene is rendered as usual, but with an extra test to see if the current fragment is in the shadow. The test is performed by projecting current vertex position into light space and using it, as texture coordinate, to sample shadow map. The result from the shadow map is then compared with z component of this projected vector, where:

- z component is a distance between the light and current fragment
- value from shadow map is distance between light and the nearest occluder

Even though shadow-mapping is suitable for real-time applications and is easily implemented, it suffers from several well-known texture-mapping artifacts, which might be difficult to get rid of. It also produces only hard shadows, which is not desired effect for rendering realistic scenes(soft-shadows are desired for purposes of realistic rendering).

Magnification artifacts occur when the projected shadow map texels cover a large area in screen space. Conversely, minification artifacts occur when several shadow-map texels are mapped to the same screen-space pixel. Solving these problems involves properly filtering the shadow map. There are many methods for reducing shadow-map aliasing, in this work Percentage-Closer Filtering and Variance Shadow Maps will be introduced.

Percentage Close Filtering(PCF)

Percentage-closer filtering [8] works by projecting the current screen-space pixel extents onto the shadow map and sampling the resulting region, a process that is similar to standard texture filtering. Each sample is then compared to a reference depth, producing a binary result. Next, these depth comparisons are combined to compute the percentage of texels in the filter region that are closer than the reference depth. This percentage is used to attenuate the light.

Increasing the size of the filter region softens the edges of the shadows. Although the quality of PCF can be very good, achieving such high quality requires a large number of samples, which leads to significant frame rate drop in real-time rendering. Figure 4.2 demonstrates soft-shadows produced via PCF method.

Figure 4.2: Shadows produced by PCF.

Variance Shadow Maps(VSM)

Another elegant solution to the problem of shadow-map filtering is to use variance shadow maps [8]. The main idea is to represent the depth data in a manner that can be filtered linearly, so that we can use algorithms and hardware that work with color and other linear data.

VSM replace the standard shadow map query with an analysis of the distribution of depth values. VSM employs variance 4.3 and Chebyshev's inequality(equation 4.5) to determine the amount of shadowing over an arbitrary filter kernel. Since it works with the distribution and not individual occlusion queries the shadow maps themselves can be pre-filtered. This allows for very fast soft shadowing with very large filter kernels.

$$M1 = E(x) = \int_{-\infty}^{\infty} xp(x)dx \tag{4.1}$$

$$M2 = E(x) = \int_{-\infty}^{\infty} x^2 p(x)dx \tag{4.2}$$

Where $M1$ and $M2$ are moments of the depth distribution recovered by filtering over a region. From these moments we can compute mean μ and variance σ^2 of distribution:

$$\mu = E(x) = M1 \tag{4.3}$$

$$\sigma^2 = E(x^2) - E(x)^2 = M2 - M1^2 \tag{4.4}$$

Using the variance, we can apply Chebyshev's inequality to compute an upper bound on the probability that the currently shaded surface (at depth t) is occluded:

$$P(x \geq t) \leq P_{max}(t) = \frac{\sigma 2}{\sigma^2 + (t - \mu)^2} \tag{4.5}$$

Variance value and probability of occluded fragment is calculated using a single texture lookup in fragment shader during the sampling of variance shadow map. The result of VSM method is demonstrated in figure 4.3.

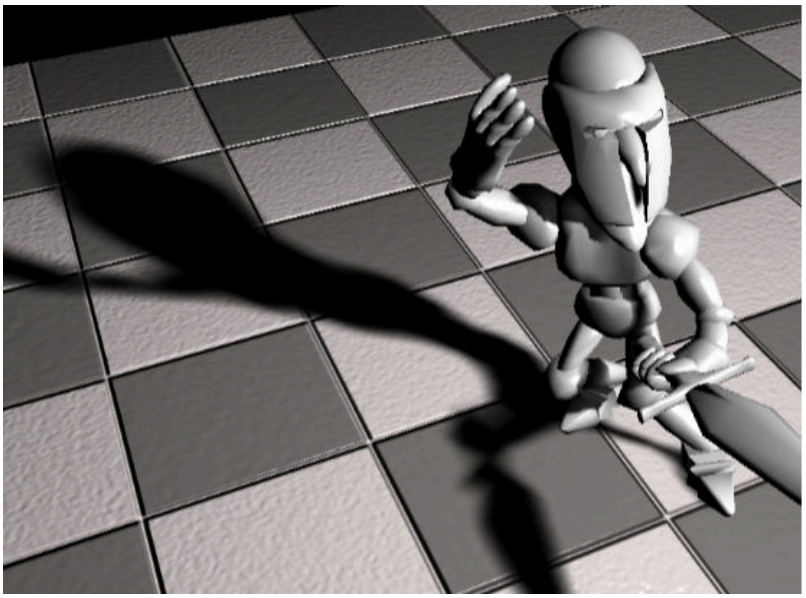


Figure 4.3: Smooth shadow produced by VSM.

4.2 Normal-mapping

Normal-mapping[19] is a technique that simulates bumpy surfaces(most real surfaces are not smooth) by modificating surface normal vectors. By modificating these normals(figure 4.4), light bounces of the surface irregulary and therefore it seems the surface is bumpy,

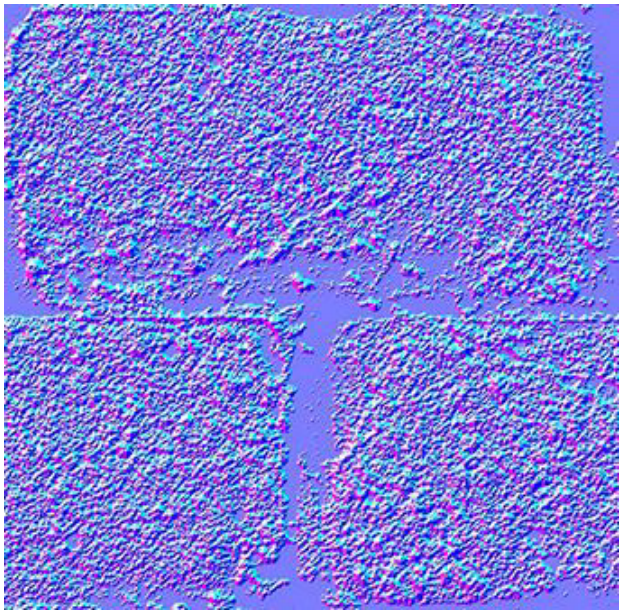


Figure 4.4: Normal vectors stored in a normal map in form of RGB components.

without need of extra geometry to describe the surfaces' irregularity(much appreciated in real-time rendering).

The information about modificated normals is stored in a normal map(figure **??**. It is a texture, which store the normal vectors in form of RGB components. [1]

4.3 Deffered shading

In computer graphics, shading means the process of rendering a lit object[6]. This process includes the following steps:

- computing geometry shape
- determining surface material characteristics(normal, diffuse color, etc.)
- calculating incident light
- computing interaction between surface and light

Typical rendering engines perform all four of these steps at one time when rendering(forward rendering) an object in the scene. Deferred shading[18] is a technique that separates the first two steps(processing geometry) from the last two(light calculations), performing each at separate discrete stages of the render pipeline.

During the process of this technique, first we create and fill so called G-buffer with geometry data. For this purpose we use frame buffer object(FBO)[2] with attachments(color attachments, depth attachment, etc.), according to, what geometric data needs to be stored.

In this step we render geometry information into created FBO attachments(textures). Most common data stored in G-Buffer are demonstrated in figure 4.5.

[1] more about normal-mapping at http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/

[2] more about FBO at https://www.opengl.org/wiki/Framebuffer_Object

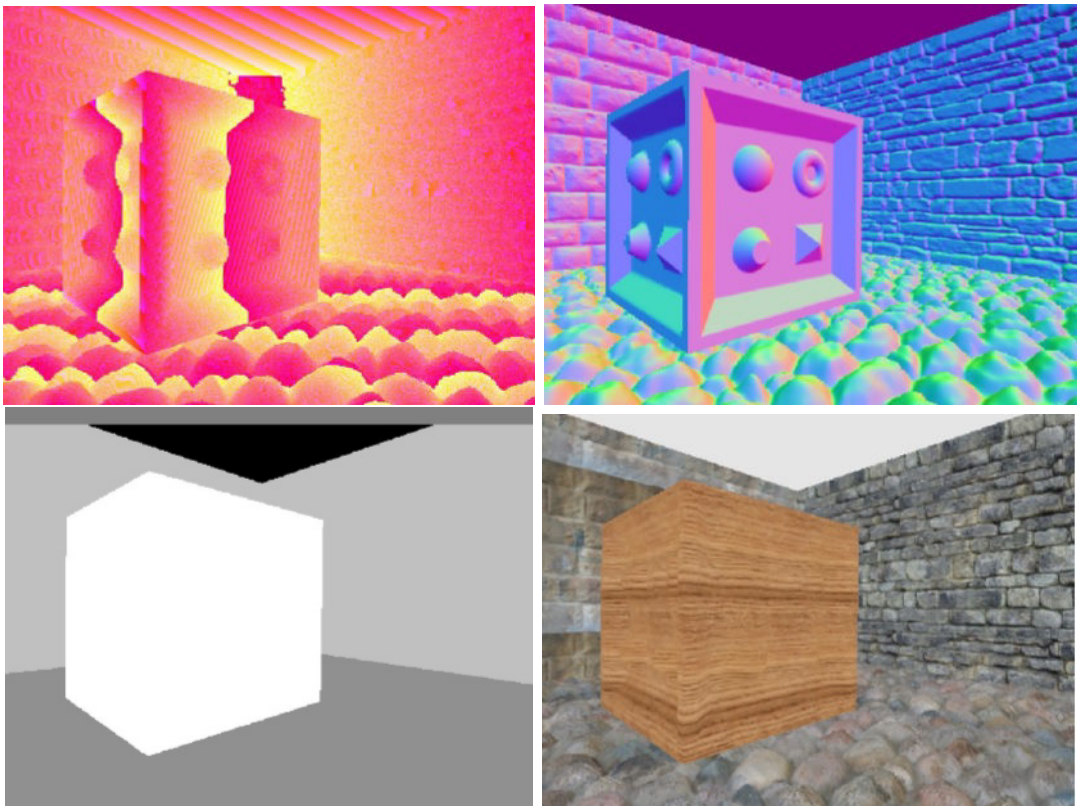


Figure 4.5: The content of G-buffer (information saved in set of textures) used in this work: **top left:** vertex positions, **top right:** normals, **bottom left:** specular color with shininess in alpha channel, **bottom right:** diffuse color.

The second step is performing light calculations. For each light, screen space bounding scissors rectangle(light is calculated only for pixels influenced by this light)[5] is calculated. For directional light full screen quad would be rendered. Information for light calcualtions is sampled from G-buffer textures.

In the process of deferred shading light is calculated only for visible pixels(visible by camera, no occluded pixels) or pixels influenced by light. Due to this fact deferred rendering offers better performace than forward rendering in scenes with many lights. It also prepares the rendering pipeline for screen space effects like screen space ambient occlusion4.4 and postprocessing effects. The main drawback of deferred shading high memory bandwidth usage, therefore format of textures storing needed data has to be chosen carefully.

4.4 Ambient occlusion

Ambient occlusion [2] is an approximation of the amount by which a point on a surface is occluded by the surrounding geometry, which affects the accessibility of that point by incoming light. In effect, ambient occlusion techniques allow the simulation of proximity shadows(the soft shadows that you see in the corners of rooms and the narrow spaces between objects). Ambient occlusion is often subtle, but will dramatically improve the visual realism of rendered scene(figure 4.6).

The basic idea behind this method is to calculate an occlusion factor for each point on

Figure 4.6: Scene rendered without ambient occlusion(left), scene rendered with ambient occlusion(right).

a surface of object and modulate lightning equation by this value(occlusion factor). More occlusion means more incoming light to a surface and vice versa. Ambient occlusion is defined as the integral of the visibility function over the hemisphere around the surface normal(figure 4.7). The most common approach to solve the integral is to cast rays over the hemisphere around each point of the surface. That is extremely expensive and leads to significant frame rate drop, therefore it is mostly used as an offline method to precompute the occlusion factor for static scenes in games and in film industry.

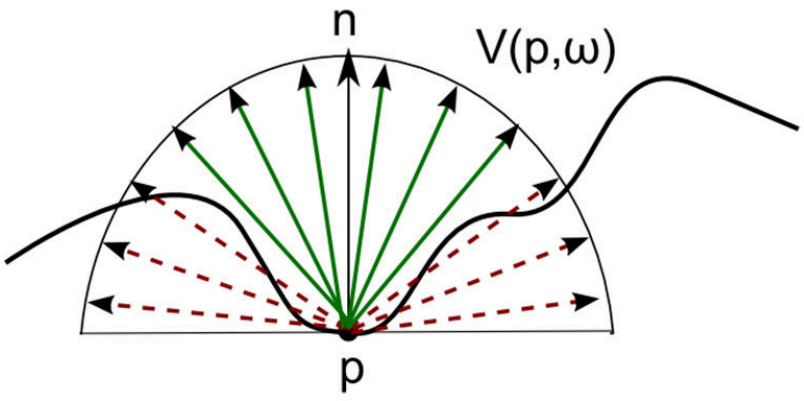


Figure 4.7: Rays casted from point on a surface of object.

For real-time applications Screen Space Ambient Occlusion(SSAO) method is used. This method was developed by Vladimir Kajalin at Crytek in 2007[10]. Rather than casting rays in a hemisphere, it samples the depth buffer at points derived from samples in a sphere(figure 4.8).

It works in the following way:

- project each sample point into screen space to get the coordinates into the depth buffer
- sample the depth buffer
- if the sample position is behind the sampled depth (inside geometry), it contributes to the occlusion factor

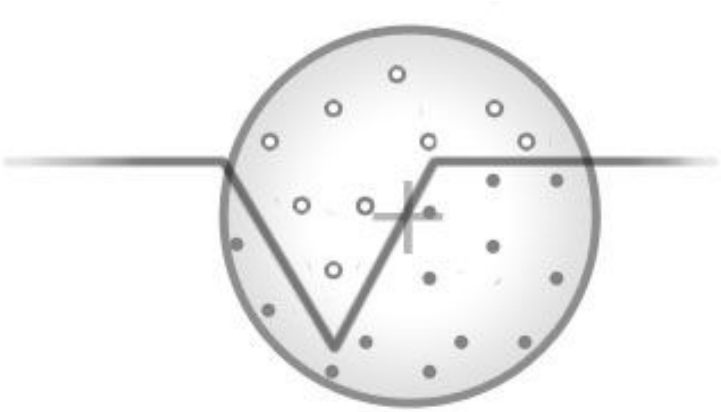


Figure 4.8: Sampled depth buffer, dots contribute to the occlusion, circles do not.

Another approach to perform SSAO[11] is to sample normal buffer and position buffer(position can be reconstructed from depth buffer). Then, the occlusion contribution of each occluder depends on two factors:

- distance d to the occludee
- angle between the occludee's N and vector from ocludee to occluder V

$$occlusion = max(0.0, dot(N, V))\dot{(}1.0/(1.0 + d)) \tag{4.6}$$

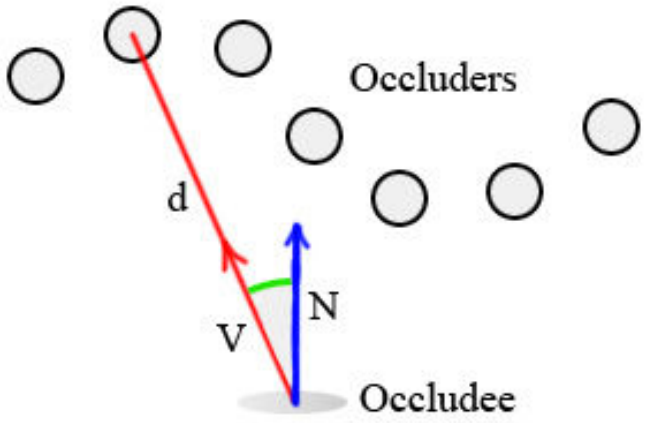


Figure 4.9: Occlusion factor calculated from the distance d between occludee and occluder and the angle between occludee's normal and the vector the the occluder.

4.5 Lens-flare

Lens-flare[1] is photographic artefact caused by internal reflections of light in lens system. Although it is an artefact, there are number of motives for simulating lens-flare fro use in computer graphics:

- it increases the perceived brightness and the apparent dynamic range of an image
- lens flare is ubiquitous in photography, hence its absence in a computer generated images can be conspicuous
- it can play an important artistic or dramatic role in image

In this section the postprocessing(the processing of the final rendered image) method to create pseudo lens-flare effect will be introduced. This method requires several full screen passes of rendered image:

- downsample the final image and filter the brightest pixels(pixels above threshold color intensity are highlighted), figure 4.10 demonstrates the result fo this step
- generate lens-flare features from downsampled and highlighted image, the process of generating features shown in figure 4.11
- blur generated features
- upscale and blend the result of lens-flare with rendered image, final result shown in figure 4.14

Figure 4.10: The result of downsampling and the brightness filtering.

The features are generated by sampling the brightest pixels along the vector from current pixel to the center of the screen.

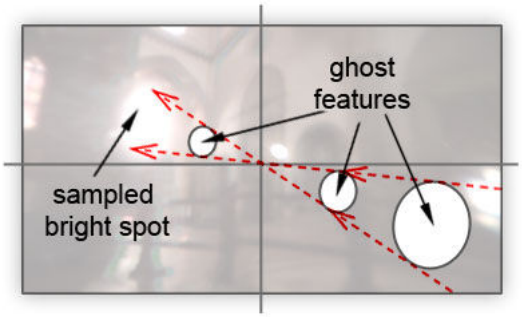


Figure 4.11: The process of generating lens features.

The generated features are then combined with sampled radial blur image 4.12.

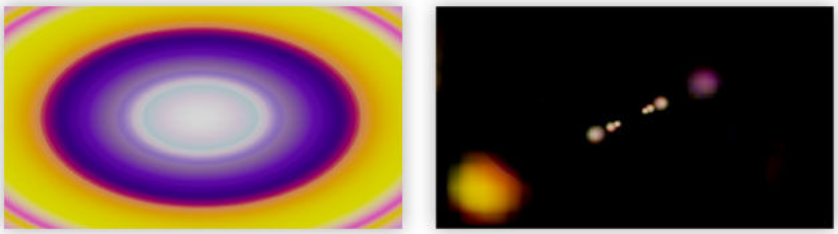


Figure 4.12: Radial blur kernel(left), the result of the radial blur kernel applied to lens features(right).

It is possible to generate „halo“ effect by modulating length of the vector from the current pixel to the centre of the screen, this leads to radially warped image. With weighting the sample to restrict the contribution of the warped image to a ring we get the result shown in figure 4.13.

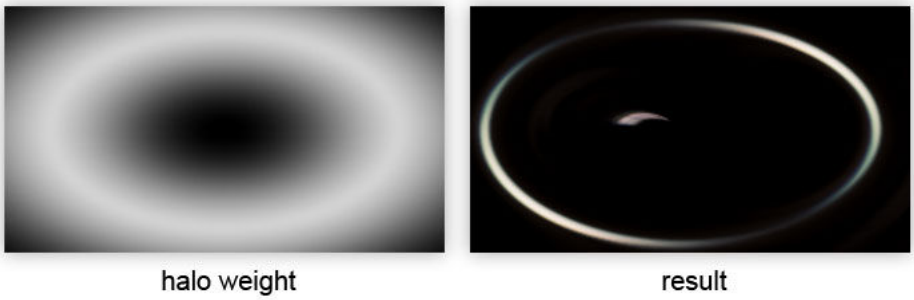


Figure 4.13: Halo effect.

As the final step generated features are blended with the original rendered image. Figure 4.14 demonstrates the result.

Figure 4.14: The result of postprocessing pseudo lens flare.

4.6 Fast Approximate Anti-Aliasing

In computer graphics rendering, there often arrises the problem of aliasing[4]. This problem is caused by discretizing a continuous signal(image). It may lead to the loss of information and undesired visual artifacts(figure 4.15).

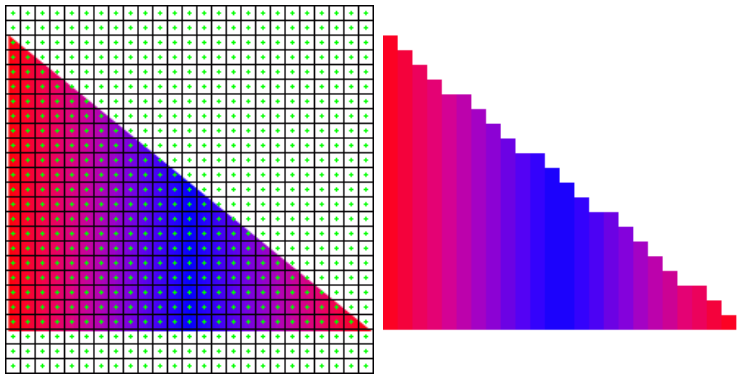


Figure 4.15: Demonstration of jagged edges.

There are many anti-aliasing methods that try to diminish the appearances of jagged

edges in real-time [3]. In this section Fast Approximate Anti-Aliasing(FXAA)[9] will be introduced.

FXAA method is a postprocessing full screen single pass method(shader pass). It works in the following way:

- in first step it converts RGB color data of the current pixel and 4 neighbouring pixels into a scalar estimate of luminance
- next it computes contrast(based on luminosity) values in x coordinates and y coordinates to get the direction of the blur, the contrast is calculated for each pixel based on 4 neighbouring pixels 4.16
- in last step it blurs the current pixel in the calculated blur direction(certain amount in positive and negative direction)

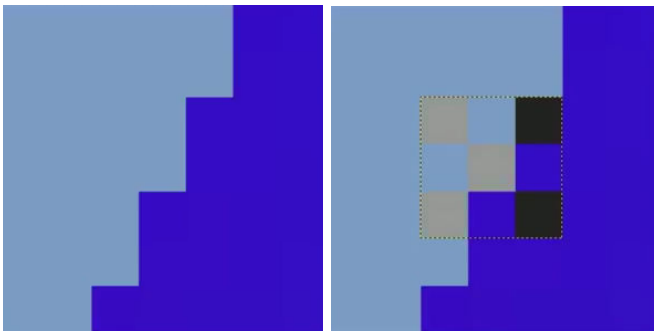


Figure 4.16: Figure demonstrats the edge detection using luminosity of 4 neigbouring pixels.

Result of Fast Approximate Anti-Aliasing is demonstrated in figure 4.17.

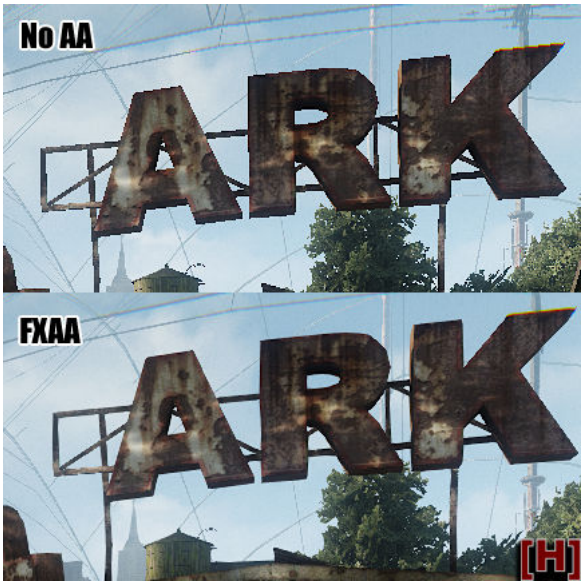


Figure 4.17: Scene rendered without FXAA(top), scene rendered with FXAA(bottom).

[3] more info at http://iryoku.com/aacourse/

4.7 Optimalizations of Real-Time Rendering

Real-time rendering of a scene with many objects(huge amount of polygons) needs to be optimized. This means there has to be a way, how to make sure, that occluded vertices or vertices out of view frustum are not sent to rendering pipeline, and do not waste computation time. There are several methods how to achieve this goal:

- view frustum culling - polygons out of view frustum of camera are culled(not sent into rendering pipeline)
- backface culling - polygons not facing the viewer are culled
- occlusion culling - polygons hidden by object closer to the camera are culled

Backface Culling

Backface culling[7] method is based on the observation that if all objects in the world are closed, then the polygons which do not face the viewer can not be seen. This directly translates to the vector angle between the direction where the viewer is facing and the normal of the polygon(if the angle is more than 90 degrees, the polygon can be discarded). Back-face culling is usually automatically performed by the rendering API(Direct3D, OpenGL) and it can be expected to cull roughly half of the polygons inside the viewing frustum.

View Frustum Culling

This method is based on the fact that only the object in view frustum should be rendered. View frustum is usually defined by six planes. Each plane, can be defined by the equation 4.7:

$$ax + by + cz + d = 0 \tag{4.7}$$

where (a,b,c) is the normal vector n of this plane and d is the distance of the plane from the coordinate system origin.

Each object in a scene is then tested against the six planes(figure 4.18). The test if a point (x, y, z) belongs to the view frustum is performed by substituting x,y,z in the plane equation 4.7. This test is then performed for all points included in bounding box(figure 4.18).

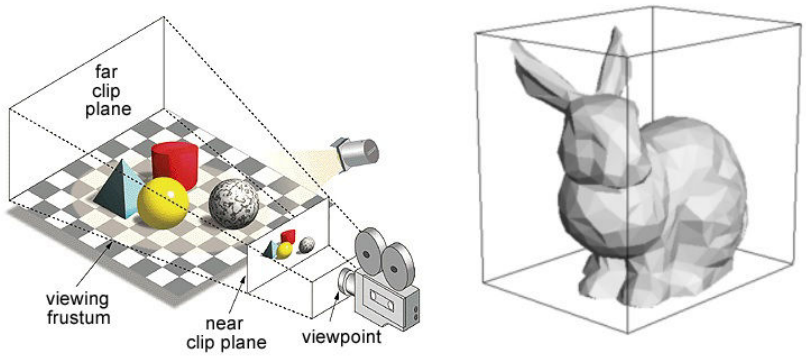


Figure 4.18: Camera view frustum with objects in it(left), bounding box of an object(right).

A problem arises when rendering complex scenes with thousands of objects. Then testing each object's bounding box against view frustum is getting very computationally expensive. Fortunately, there is a better way based on hierarchical subdividision of space into smaller chunks, and tagging which objects belong to which chunks. One of the methods(Octree) for space partitioning is described in the section 4.7.

Occlusion Culling

Occlusion culling[15] increases rendering performance simply by not rendering geometry that is outside the view frustum or hidden by objects closer to the camera. Two common types of occlusion culling are occlusion query and early-z rejection.

Occlusion query

Occlusion queries count the number of fragments that pass the depth test, which is useful to determine visibility of objects. If an object is drawn but zero fragments passed the depth test, it is fully occluded by another object. In practice this means that a simplification of an object is drawn using an occlusion query (a bounding box can be the occlusion substitute for an object) and only if fragments of the simple object pass the depth test, the complex object is drawn[4].

Early-z Rejection

In the rasterizer stage of the rendering process, early-z compares the depth value of a fragment to be rendered against the value currently stored in the z-buffer. If the fragment is not visible (because the depth test failed), rejection occurs without fetching the texture for the fragment or executing the fragment program. This results in memory bandwidth being saved at the per-fragment level.

Octree Space Partitioning

An octree[12] is a data structure used for 3D space partitioning(also for collision detection).

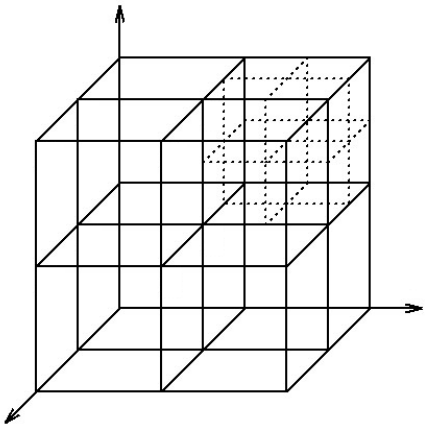


Figure 4.19: Demonstration of subdivided octree.

Space partitioning using octree works in a following way:

[4]more about occlusion queries at http://http.developer.nvidia.com/GPUGems/gpugems_ch29.html

- calculate dimensions(width and height) of a scene, create root node(bounding box of calculated dimensions), calculate polygons in root(test polygons against four planes of the bounding box)
- recursively subdivide each node(starting with root) into eight new nodes(half dimension of the parent node) if number of polygons exceeds threshold(maximum amount of polygons allowed in a node), and maximum depth was not reached
- if number of polygons of current nodes is lower than treshold, store polygons in current node

Figure 4.20 demonstrates a subdivision of an octree, an octree is subdivided only in nodes with geometry exceeding the poylgon count threshold.

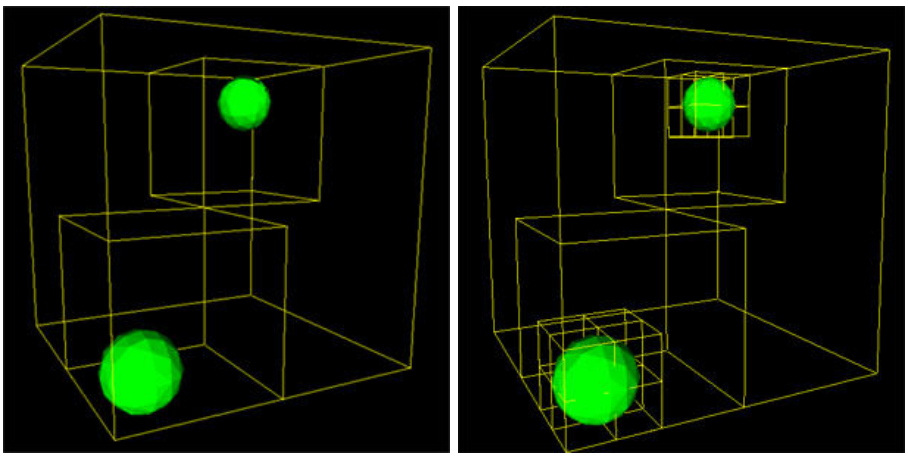


Figure 4.20: Octree subdivided only in nodes including the geometry.

After the octree is created(once before the rendering, this way not applicable to dynamic scenes) and its nodes filled with the polygons, the octree is rendered by recursively passing through the nodes, where node's bounding box is checked against the view frustum, if it passes the test and the node is not subdivided, we render stored geometry in the node, if the node is subdivided we continue with the recursion.

This way a scene geometry is culled by view frustum with the expense of searching in octree structure. When designing an octree, maximum depth and maximum polygon count has to be chosen wisely(recursion might become inefficient for interactive frame rates).

Chapter 5

Design and Realization of Real-Time Rendering Engine

The aim of this chapter is to describe methods and techniques used, to implement the application for real-time rendering, simulating and recording of scenes with walking pedestrians.

5.1 Models and Skeletal Animation

At first, implemented engine needs to load created models(geometry, textures, etc.) along with information for animations, when loading character models. For this purposes I used ASSIMP[1] library which allows loading of all needed information in several file formats. It loads data into its own internal data structures that are then available for further processing. The main data processing includes:

- reading materials along with their textures(diffuse, specular, normal, etc.), loading textures to gpu
- reading the geometry data, creating and filling particular vertex buffer objects
- reading the info about what bones(bones together create skeleton), and how, influence what vertices, according to this, vertices are transformed in vertex shader 5.1
- reading animations, this includes interpolating between frames of animation(playing the animation, blending between two animations, etc.), in each frame of rendering, final transformation matrix of each bone is calculated
- etc.

The following classes implement loading and processsing of models and animations:

- Model.cpp - takes care of loading models, each model can be divided into several meshes(according to the count of used materials - this division is a part of ASSIMP internal functionality)
- Mesh.cpp - represents a single part of a model, stores information about vertices, indices, material that this part is made off and creates a vertex array object with vertex array buffers(vertices, indices, tangents, bones, etc.)

[1] for more info visit http://assimp.sourceforge.net/

- Animation.cpp - this class takes care of loading animations, interpolating between their frames to play the animations, or blend between two animations

Listing 5.1: Animated vertex transformed in vertex shader, number of bones influencing a vertex limited to 4

```
...
layout (location = 4) in vec4 Weights;
uniform mat4 gBones[MAX_BONES];
...
mat4 BoneTransform = gBones[BoneIDs[0]] * Weights[0];
BoneTransform  += gBones[BoneIDs[1]] * Weights[1];
BoneTransform  += gBones[BoneIDs[2]] * Weights[2];
BoneTransform  += gBones[BoneIDs[3]] * Weights[3];
vec4 PosL = BoneTransform * vec4(Position, 1.0);
gl_Position = projMat*viewMat*modelMat*PosL;
...
```

For the demonstration purposes, I created a scene in Blender with a character and its animations for walk and idle state(figure 5.1).

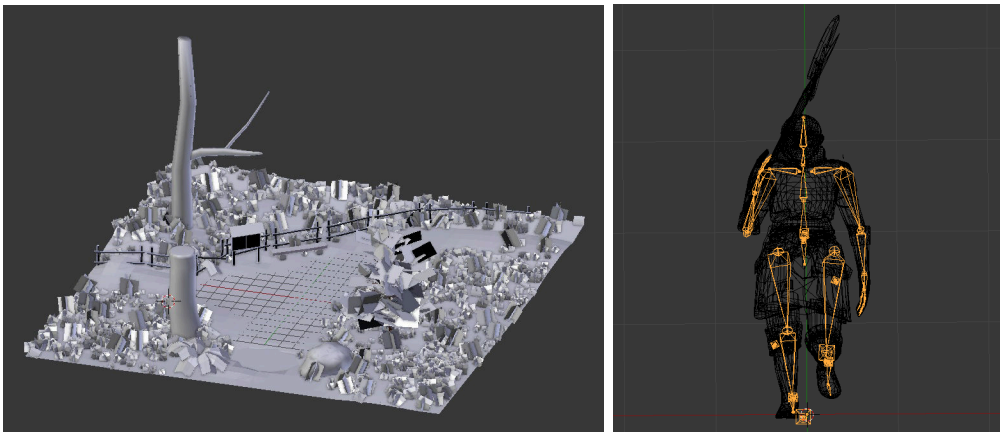


Figure 5.1: A scene created in Blender(left), rigged and animated character(right).

5.2 Artificial Intelligence for Character Movement

For character movement I chose the grid map method(described in section 3.2). The grid map is created by rendering the scene and its objects from top view, in orthographic projection. This way a raster image is created with non-zero pixels representing static obstacles(figure 5.3). Then the capability of the map is calculated and A* algorithm searches this map to find the paths, which characters should be going along(they avoid static obstacles).

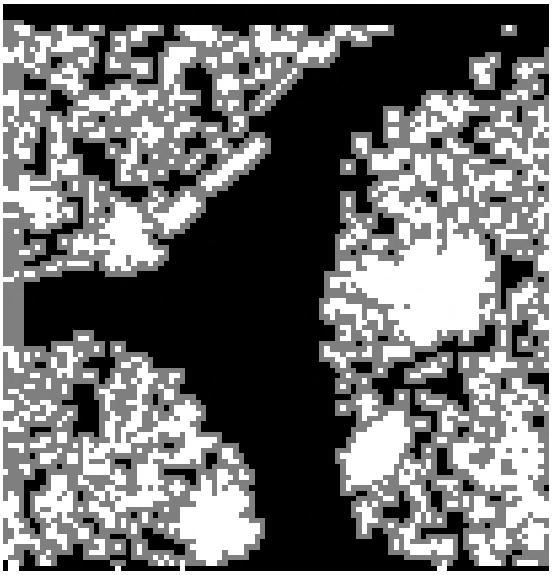


Figure 5.2: The grid map representation of a scene with static obstacles.

Of course avoiding static obstacles is not enough to achieve realistic movement, the characters need to avoid dynamic obstacles too(in this case each other). Also the movement of the characters, due to the fact that grid map is used, might seem a bit unrealistic(zig-zag movement). I resolved both zig-zag movement and avoiding dynamic obstacles by implementing steering behaviors(section 3.3):

- seek behavior - characters move smoothly(when appropriate road radius is set) towards their targets
- separation behavior - characters keep distance between each other
- path follow behavior - characters try to stay in road radius, when they reach out of it they are pushed back into it
- evade behavior - characters look ahead, when they see obstacles at some distance(depending on see_ahead vector and its intersection with radius of other character), they tend to move away from it(avoid it)

The following classes implement methods and techniques used to achieve character movement:

- Character.cpp - represents a character model, calculates the movement using steering behaviors and moves the character
- Group.cpp - characters are gather into groups, each group has start and its goal and the path between them, groups aslo describe how should characters behave in them
- Map.cpp - represents a map, stores raster image as a grid map representation, finds the path(path of particular group) in this grid map and calculates the capability of the map

Figure 5.3: Characters moving along the calculated paths, the road radius(green circles) indicates the radius a character needs to get into, before moving to the next target(as a part of seeking behavior, movement is smoothed with bigger radiuses), blue areas indicate a character's width and it is used in the separation behavior, so the characters keep distance between each other.

5.3 Rendering Pipeline

For rendering I implemented deferred renderer(section 4.3) using octree data structure(4.7) for geometry optimalization. The rendering is happening in 3 passes:

- in the first pass the scene is rendered into variance shadow map(section 4.1)
- the second pass creates G-buffer(section 4.3) with textures for normals, positions, specular color(shininess in alpha channel) and diffuse color
- the third pass then reads G-buffer textures and performs the lighting calculations in a form of phong lighting model, normal-mapping(section 4.2), screen space ambient occlusion(section 4.4) and variance shadow-mapping(section 4.1)

After rendering stage I implemented postprocessing effects in a form of lens-flare(section 4.5) and fast approximate anti-aliasing(section 4.6).

The following classes participate in the rendering and the postprocessing stages:

- GLMiniWidget.cpp - used for rendering the contexts of model previews
- GLWidget.cpp - loads scene models, renders a scene, performs simulation
- MyShaders.cpp - compiles and links shader programs
- MyVertex.cpp - saves all info needed about vertex(normal, texture coordinates, tangent, info about bones that influence this vertex, etc.)
- Octree.cpp - subdivides spaces so the geometry may be culled against the view frustum in interactive frame rates
- Texture.cpp - manipulates the textures(loading from file, loading to GPU, etc.) and can serve as frame buffer object(FBO)

I implemented all the methods and techniques in rendering pipeline using GLSL[14] shader programs:

- sm.vs, sm.fs - shader programs for variance shadow-mapping
- gbuffer.vs, .fs - creating G-buffer, storing data into textures
- deferred.vs, .fs - reading G-buffer, lighting calculations, SSAO, VSM
- lens_feature.vs, .fs - creating ghost features for lens-flare
- threshold.vs, .fs - highlighting the brightest pixels
- add_shader.vs, .fs - mix lens features with original image
- fxaa.vs, .fs - fast approximate anti-aliasing, uses luminosity for edge detection
- minigl.vs, .fs - shader used for rendering previews of models(only phong lighting model), used in forward rendering
- skybox.vs, .fs - shader for rendering skybox or environment(skybox is allways in far plane of the view frustum, environment is not moving with camera)

Figure 5.4 demonstrates rendered scene with animated characters and implemented effects.

Figure 5.4: The result of the rendering and postprocessing stages.

5.4 Graphical User Interface(GUI)

I implemented GUI using Qt[2], so the user can customize own scenes, create own movement of the characters, edit visualization of a scene(customizing SSAO, lens-flare, etc.), record simulated scene, etc. The functionality of the GUI can be divided into three main categories(described by the following figures):

[2]more info at http://www.qt.io/developers/

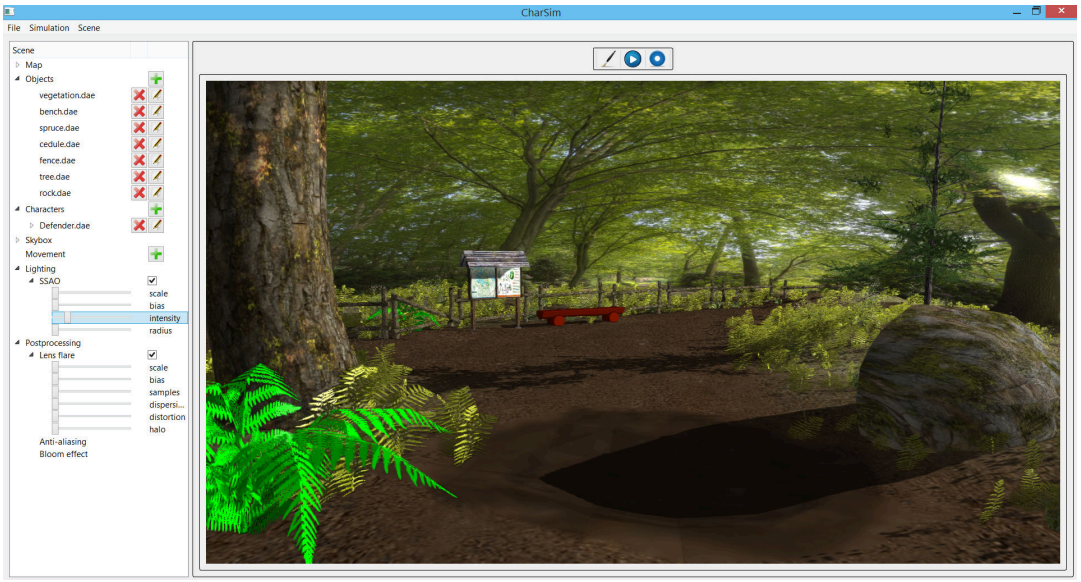


Figure 5.5: User is able to load a scene, add/delete models, characters and is able to customize vizualization of a scene.

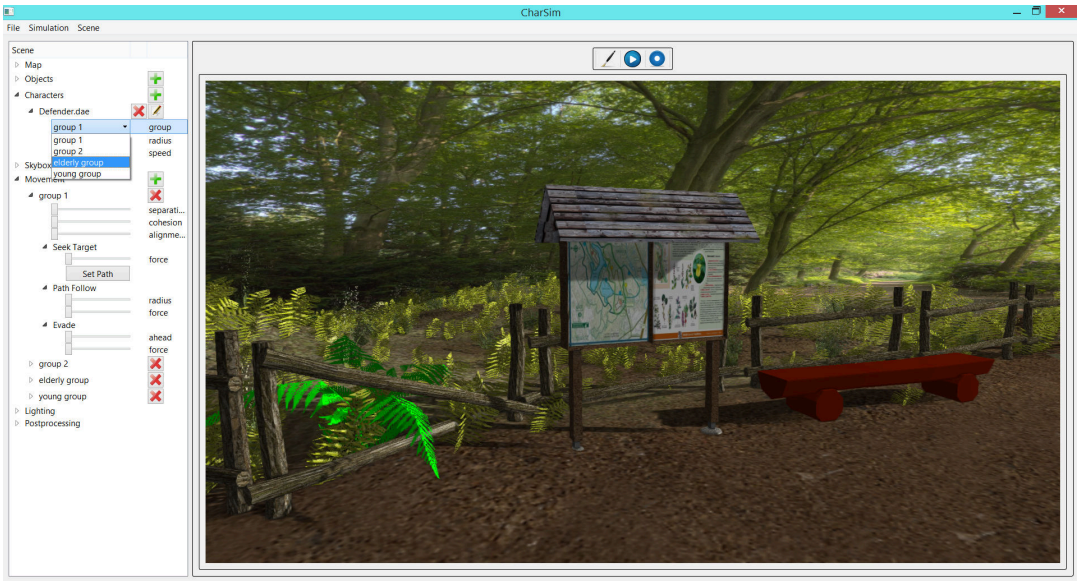


Figure 5.6: asdasdasd

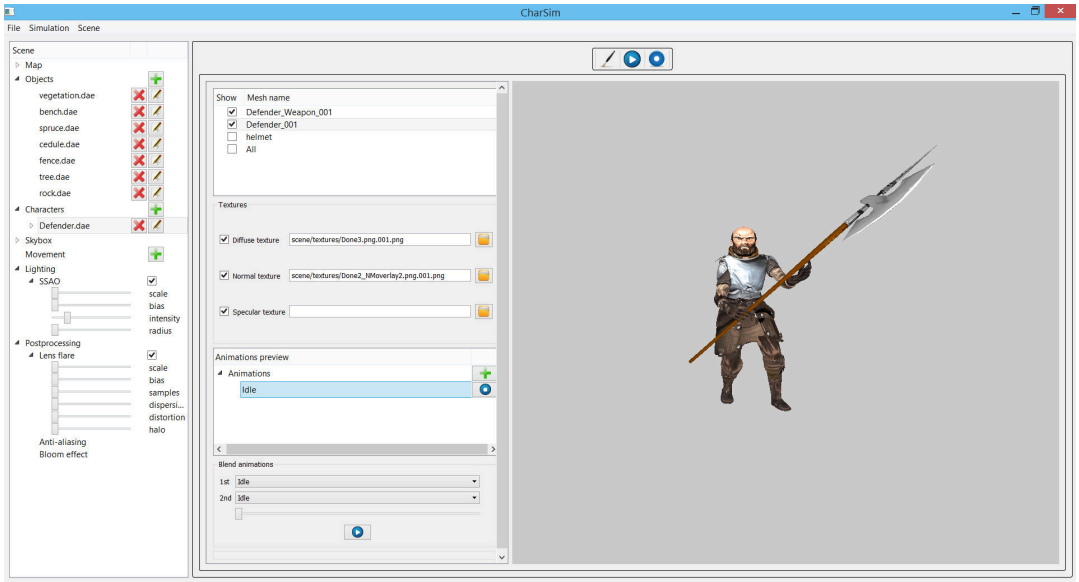


Figure 5.7: User is allowed to preview loaded models, change/activate/deactivate textures(applies to the environment texture too), load and play/blend animations, customize what parts of the model will be displayed(can be used for clothing of the character).

Chapter 6

Results and Use-cases of Implemented Engine

This chapter demonstrates the results, performance and the controls of implemented engine, tested on hardware:

- CPU: AMD FX(tm)-6300 Six-Core Processor 3.50 GHz
- RAM: 12 GB DDR3 1600 MHz
- GPU: ATI Gigabyte HD 7870 2 GB DDR5

6.1 Results and Performance

Screen Space Ambient Occlusion

The performance and the results of SSAO technique are demonstrated in the figures(figure 6.1a without SSAO and figure 6.1b with SSAO).

The cost of screen space ambient occlusion is approximately 4 frames per second. This cost is eligible for real-time engine because of the realism it brings to the scene(the effect might be more visible from video presentation in section(7)).

Fast Approximate Anti-Aliasing

Another technique that brings a lot of realism to the scene is FXAA(fast approximate anti-aliasing), it is used for smoothing the jagged edges(caused by discretization of input data). The cost of this technique tested in implemented engine is approximately 1.5 ms(lower than 1 frame per second) for 1920x1080 resolution. The cost and the results that FXAA offers makes it really powerful anti-aliasing technique. The results are demonstrated in figures 6.2a and 6.2b.

Variance Shadow-Mapping

Shadows are important part of 3D computer generated graphics. For shadows I implemented variance shadow-mapping that generates smooth soft-shadows. The cost of this technique in comparison to simple shadow-mapping(hard shadows) is nearly the same(approximately 1 frame per second slower), and it offers much better results(also solves some of the artifacts of simple shadow-mapping).

Octree Space Partitioning

The geometry of rendered scene is culled against view frustum. Testing bounding box of all objects in the scene against view frustum is not appropriate for real-time rendering. Therefore the geometry(space) is subdivided into chunks using octree data structure. This significantly increases the rendering frame rates. The figures 6.4,6.5 and 6.6 demonstrate view frustum culling using space partitioning and the improvement it brings to the frame rates.

6.2 How to Run and Control the Application

The implemented application runs under Qt. It uses following libraries:

- GLEW - shaders
- OpenGL - rendering
- DevIL - textures
- ASSIMP - model loading
- FFmpeg - recording

Use-case Example

To simulate the scene with walking pedestrians, after running the aplication(from Qt Creator for example), there is need to load a scene(File → Open → last scene, to add the movement group/groups(Movement tab on the left side of the GUI → the add button next to it). After adding the group/groups you can assign the characters to the groups(open Characters list and the list of the properties of particular character under it). Using the edit button you can get to the preview of the model of the character and its animations(add animation for walk, so the character is not standing in place in idle state). After setting this, you can get back to rendering a scene, with the brush button in the tool bar, above the preview of the model. Then, it is needed to set the starting and the ending point of each group(Movement → Group → Seek Target → Set Path button + selecting two points from the scene). Then you can start the simulation by clicking on the play button in the toolbar above the rendered scene(the paths get calculated and the characters instanced). If everything set correctly you can adjust the forces in the group by available sliders to create desired movement of the characters, you can use sliders in Lighting and Postprocessing sections to adjust the visualization of the effects.

Control of the application

- camera rotation - hold the middle mouse button
- camera zoom - the middle mouse scroll
- camera shift - shift+hold the middle mouse button
- a - activates/deactivates FXAA
- h - hides the paths and visualization of forces

- o - displays/hides the nodes of the octree
- e - switches between skybox and environment
- z - switches between wireframe and full render mode
- s+middle mouse hold - scales selected character(must be selected in the left panel of the GUI)

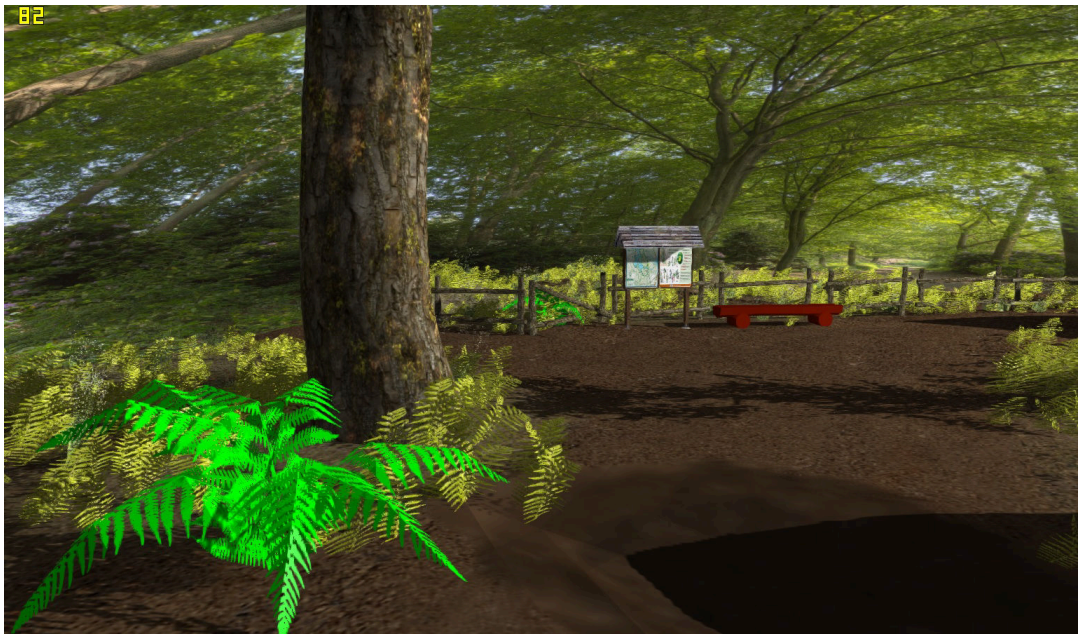
Files and Folders

- scene/characters - folder with the files of the character models
- scene/animations - folder including the animations of the characters
- scene/textures - textures for the models of the scene
- scene - this folder includes the models of the scene and other models needed for viual representation of forces, etc.
- shaders - folder with the implementations of the shader programs
- screenshots - results of the rendering and simulation
- icons - icons used by the GUI
- . - source and header codes, the recorded video is saved here as output.mp4

Demonstration Video

The following video was used at Excel@FIT conference that took place on April 30 2015(space partitioning and variance shadow-mapping were not implemented):

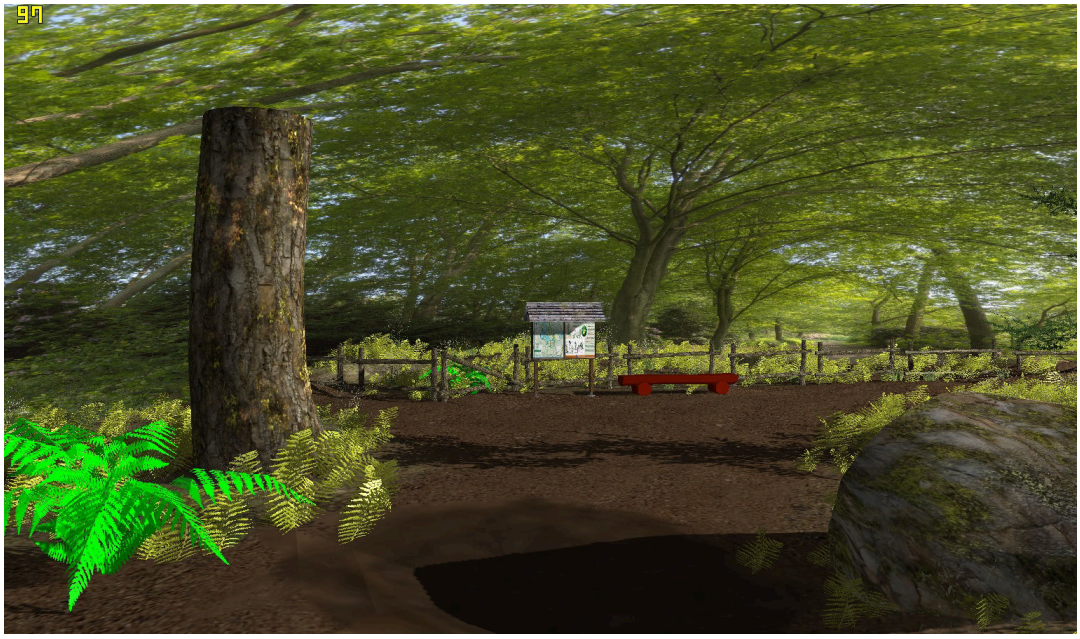
- https://www.youtube.com/watch?v=2bQV2ftENdc



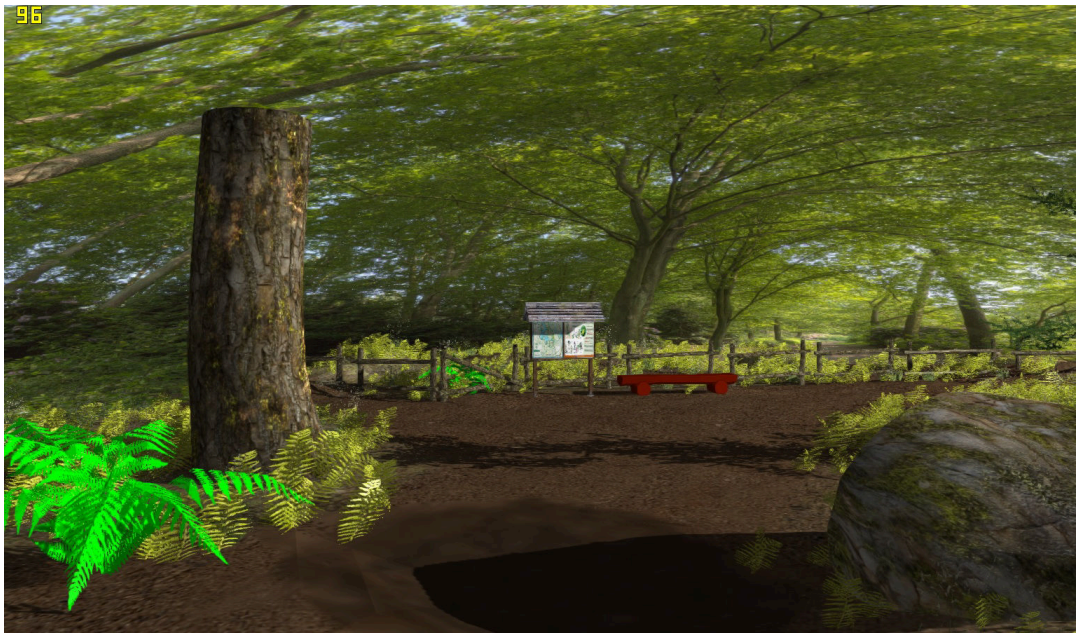
(a) Scene rendered without SSAO.

(b) Scene rendered with SSAO.

Figure 6.1: SSAO results



(a) The scene rendered without FXAA.



(b) The scene rendered with FXAA.

Figure 6.2: FXAA results

(a) The shadows using simple shadow-mapping.

(b) The shadows using variance shadow-mapping.

Figure 6.3: Comparison of simple shadow-mapping and variance shadow-mapping.

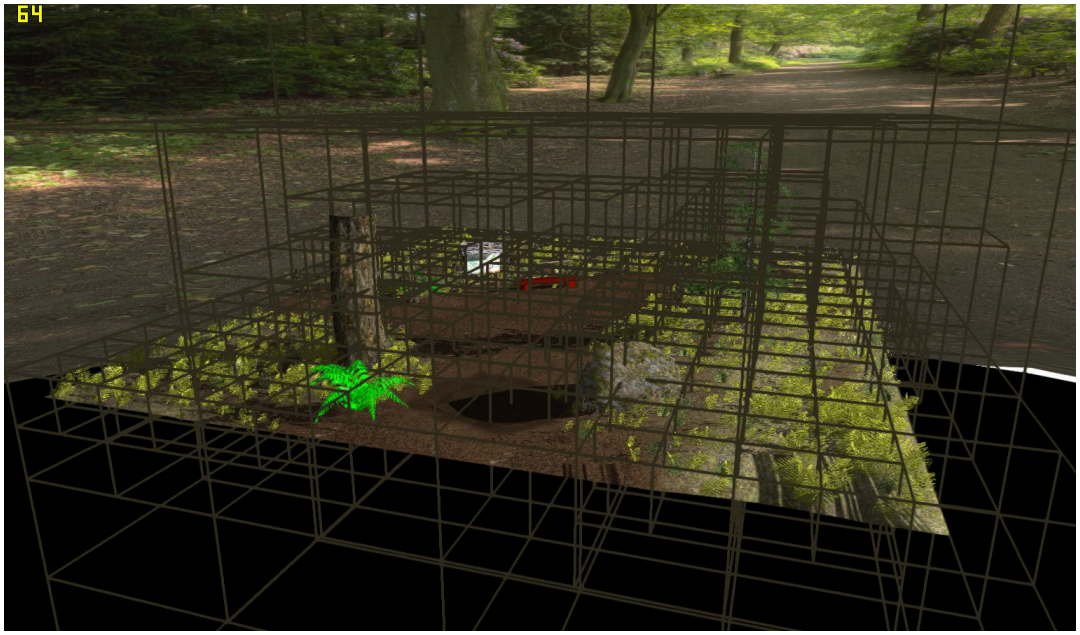


Figure 6.4: All geometry of the scene rendered(22 216/22 216 traingles). Maximum level of octree set to 4, maximum triangles in node set to 100.

Figure 6.5: Camera placed in the scene, 10 082/22 216 triangles rendered.

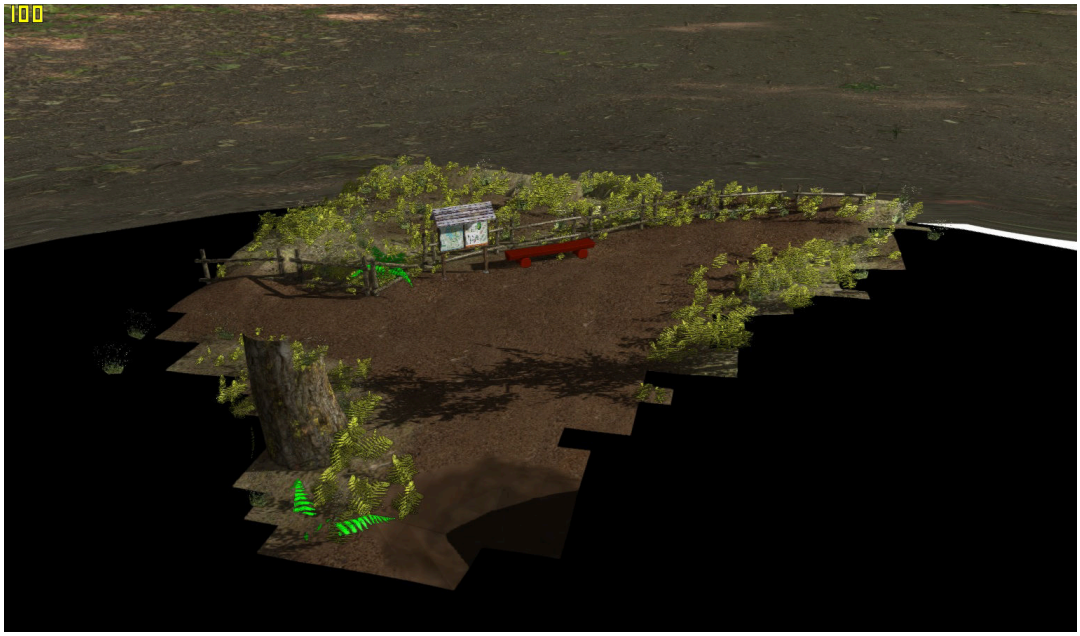


Figure 6.6: Scene culled against view frustum culling of the camera.

Figure 6.7: Simulated scene with 24 pedestrians(runs smoothly in 65 frames per second).

Figure 6.8: Simulated scene with over 40 pedestrians. This scene runs in 4 frames per second, it is mainly caused by the geometry of the character(around 9000 vertices per character, if simulating with more low-poly characters, the performance gets better). Generating images even in this low frame rate is suitable as eventual test data input for people counting or similar systems.

Chapter 7

Conclusion

In this master thesis I succesfully implemented an application(engine) able to load, render, simulate and record various scenes with walking pedestrians. I embedded the engine's functionality into Qt GUI. User can customize own scenes, adjust the visualization of these scenes(customizing shader effects), the visualization of loaded models(change textures, display different parts of the model, etc.). User is allowed to set the movement patterns of the pedestrians by setting the paths(path-finding) and adjusting the behaviors of the characters(steering behaviors).

Due to the fact that the problem, this thesis solves is quite extensive, I did not manage to implement all the goals I had when planning the work. In the future work I would like to finish the work concerning:

- characters moving using navigation mesh 3.2
- serialization of the objects(XML), so the scenes and other settings can be saved and loaded back
- optimalizations including dynamic scenes(implementing dynamic space partitioning data structure) and level of detail
- general tuning up the GUI, sometimes the application is unstable, adding more functionality(setting number of instances of the characters, parametrizing FXAA, more postprocessing effects like bloom effect, etc.)

Bibliography

[1] John Chapman. Pseudo lens flare. *john-chapman-graphics*, 2013.

[2] John Chapman. Ssao tutorial. *john-chapman-graphics*, 2013.

[3] COLLADA.org. Collada - digital asset and fx exchange schema [online]. `https://collada.org/`, 2014 [cit. 2015-01-03].

[4] Franklin C. Crow. The aliasing problem in computer-generated shaded images. *The University of Texas at Austin*, 1977.

[5] Francisco Fonseca Fabio Policarpo. Deferred shading tutorial. *Pontifical Catholic University of Rio de Janeiro*, 2007.

[6] Rusty Koonce. *GPU Gems 3*, chapter Deferred Shading in Tabula Rasa. Pearson Education, Inc., 2007.

[7] Pietari Laurila. Geometry culling in 3d engines. *Graphics Programming and Theory*, 2000.

[8] Andrew Lauritzen. *GPU Gems 3*, chapter Summed Area Variance Shadow Maps. Pearson Education, Inc., 2007.

[9] Timothy Lottes. Fxaa. *ACM SIGGRAPH 2011 Course*, 2011.

[10] Martin Mittring. Finding next gen - cryengine 2. *Advanced Real-Time Rendering in 3D Graphics and Games*, 2007.

[11] José María Méndez. A simple and practical approach to ssao. *Graphics Programming and Theory*, 2010.

[12] Eric Nevala. Introduction to octrees. *Game Programming*, 2014.

[13] Amit Patel. Path-finding. *Red Blob Games*, 2014.

[14] Jr. Richar S. Wright, Nicholas Haemel, Graham Sellers, and Benjamin Lipchak. *OpenGL superbible*. Addison-Wesley BOSTON, 2010. ISBN 978-0-321-71261-5.

[15] Dean Sekulic. *GPU Gems*, chapter Efficient Occlusion Culling. Pearson Education, Inc., 2007.

[16] Anirudh.S Shastry. Soft-edged shadows. *Graphics Programming and Theory*, 2005.

[17] Daniel Shiffman. *The Nature of Code*. 2012. ISBN: 0985930802.

[18] Oles Shishkovtsov. *GPU Gems 2*, chapter Deferred Shading in S.T.A.L.K.E.R. Pearson Education, Inc., 2005.

[19] Naty Hoffman Tomas Akenine-Moler, Eric Haines. *Real Time Rendering*. A K Peters, Ltd., 2008. ISBN 978-1-56881-424-7.

[20] Steven White. *Game Programming Gems 3*, chapter A Fast Approach to Navigation Meshes. Charles River Media, Inc., 2002.