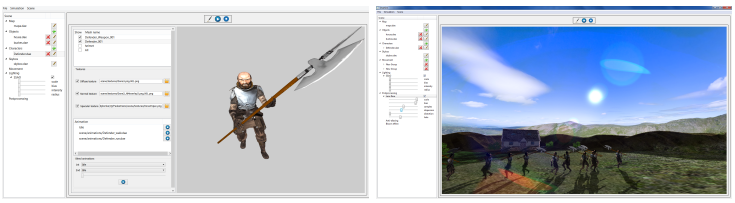
Excel@FIT 2015
http://excel.fit.vutbr.cz

Real-Time Rendering of a Scene With Many Pedestrians

Václav Pfudl



Abstract
The aim of this text was to describe implementation of software that would be able to simulate a scene with walking characters in real-time with emphasis on rendering level of realism. The application should be then able to record video sequences of such simulated scenes. Such video sequences could be used as an input (test data) for people counting systems. 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 creating a model of the character and using skeletal animation. To achieve 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 rendering methods such as normal-mapping, shadow-mapping, deffered rendering, skydome, lens flare effect and screen space ambient occlusion. Rendering stage of a scene can be easily parametrized through implemented GUI. Implemented application provides user with easy way of setting a scene with walking pedestrians, setting its visualization and to record the result.

Keywords: real-time rendering, normal-mapping, shadow-mapping, lens-flare, screen space ambient occlusion, skydome, skeletal animation, movement artificial intelligence, group behaviors

Supplementary Material: Demonstration Video

*xpfudl01@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

The motivation behind this paper was to develop an application that would be able to record video sequences of scenes with walking chracters in high level of realism so the video sequences could be used as an input (test data) for people counting systems. The developed application itself could serve as a tool for other purposes like making preanimated scenes for games and movies. The desired output of such application should be a program with graphical user interface which would allow users to parametrize visualization of a scene (scene made of imported models) as well as customize characters movement according to required simulation. The output should be recorded with higher frame rates, althought overall quality of visualization is preferred over frame rates.

The development of such application was divided into three major groups: character animations, artificial intelligence for movement of characters and advanced real-time rendering techniques.

When rendering a scene with walking pedestrians, first that scene needs to be populated with animated character models. Blender [1] was used to create such

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

models (not only character models but also static scene models) along with their animations (skeletal animation, section 2.1). Then we need a tool to import geometry, animations and material data of these models into our application. When choosing which tool to use we need to consider which file formats this tool can work with along with what that certain file format can describe (animations, textures, materials, lights etc). A convenient tool for our purposes is ASSIMP (Open Asset Import Library) along with COLLADA file format.

Each character needs to act autonomously, how to get to their desired location and react dynamically to the surrounding environment (avoid static and dynamic obstacles). Most of path finding algorithms are based on Dijkstra's algorithm [1]. For most cases A* algorithm [2] is the one most effective and our case is not different. Our agent (character) needs to percieve its environment to avoid other characters or to group up with other characters etc. In modern game engines or in applications simulating group behavior, algorithms are based on algorithmic steering behaviors for animated characters, which were developed by Craig Reynolds in late 1980s [3, p. 261-300].

Considering visualization of whole scene, at the moment there are two main 3D rendering API including DirectX and OpenGL. For the sake of our application we use OpenGL mainly becasue it is multiplatform API. To visualize scene in more realistic manner we need to program how rendering pipeline will process vertices and fragments of our models we export from Blender and import to our application. For these purposes GLSL (OpenGL shading language) is available, which is the language that allow us to program various stages of rendering pipeline including vertex shaders, tesselation shader, geometry shaders, fragment shaders etc. This allows to produce some advanced shader effects and even postprocess rendered output (bloom effect, antialiasing, lens flare etc.).

Our application successfully encapsulates above described functionality into graphical user interface that provides users to set various scenarios with walking pedestrians, enables to set group behaviors of these pedestrians along with how the whole scene will be visualized (which shader effects scene uses, lighting of the scene etc.). Application also allows to record these animated scenes.

2. Moving characters around

Our implemented application includes the following methods and techniques ensuring autonomous movement of character in dynamic environment.

2.1 Skeletal animation

To animate model of a character skeleton of this model (this skeleton is commonly called rig) was created in Blender. Each bone of this skeleton is then assigned with vertices this bone should have control over (these vertices are weighted, figure 1). Then set of frames is created and each frame carries information of a pose of the model in the way it makes consecutive movement (walking, running etc.) after interpolating between frames. This information of bones, influenced vertices and animations is then imported into application using ASSIMP library and model is animated in vertex shader program. In each rendered frame we calculate current pose of each bone (transformation matrix) of character's skeleton and then we multiply this matrix with vertices according to which bone they belong to.

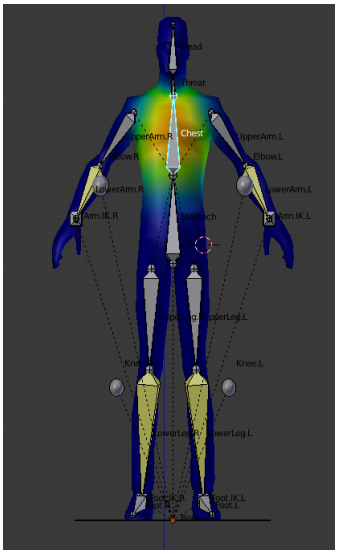


Figure 1. This figure demonstrates weighted skeleton created in Blender, particulary how vertices around chest are influenced by "chest" bone.

2.2 Artificial intelligence for character movement

As the pathfinding algorithm we chose A* algorithm for its efficiency and flexible use in wide range of contexts. Map representation we chose is demonstrated in figure 2. Navigation mesh [4] allows characters move much smoother without "zig-zag" movement (in context of a triangular area character can move in straight line) and it also solves static obstacle avoidance. To make sure characters act autonomously steering behaviors [3]. Steering behaviors allow characters to percieve their surrounding environment (position vectors, acceleration vectors, etc. are available) and act to avoid obstacles or collisions with other characters by applying vector forces to their current acceleration. Characters in our application are divided into groups

and each group can be parametrized to set behaviors of characters in that particular group. Behaviors are vector forces applied and calculated for every single character.

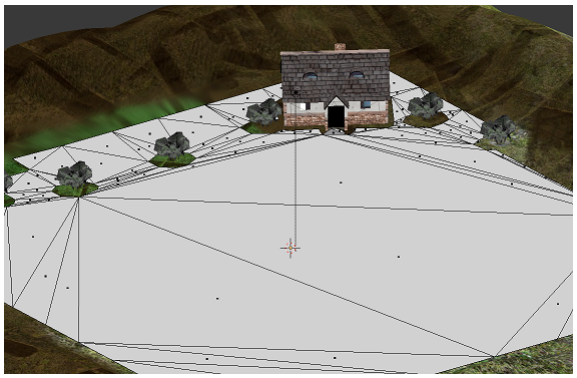


Figure 2. Automatically generated navigation mesh [4] in Blender. In triangular areas characters can move in straight lines (no need to calculate path), edges serve as portals into other triangular areas and vertices are nodes which we search paths in (in case we can't move in straight line to our target).

3. Realistic scene rendering

In order to visualize scene with higher level of realism and to render scene in real-time (more than 25 frames per second) we implemented following techniques.

3.1 Normal-mapping

Rendering scene with Phong lighting model interpolates light nicely and conveys a sense of desired realism, however it can not simulate bumpy surfaces without use of high amount of extra vertices (inappropriate for real-time). Therefore we used normal-mapping [5] technique applied to low-poly (model composed of low amount of polygons) models. Using this technique is suitable for real-time applications because at run time it costs one texture lookup more than simple Phong model and offers big amount of detail contributing to realistic look of models. Figure 3 shows the impact of normal-mapping used on low-poly model of a character.

3.2 Shadow-mapping

To aproximate realism we need objects to cast shadows. To achieve this effect we use shadow-mapping [6]. This technique itself requires two pass rendering which means we have to render our scene twice. This would imply that we get frame rate dropped by half, however during first pass we only need to right down a depth information of vertices and there is no need to deal with lighting calculations. This makes this technique viable for real-time shadow casting. This technique

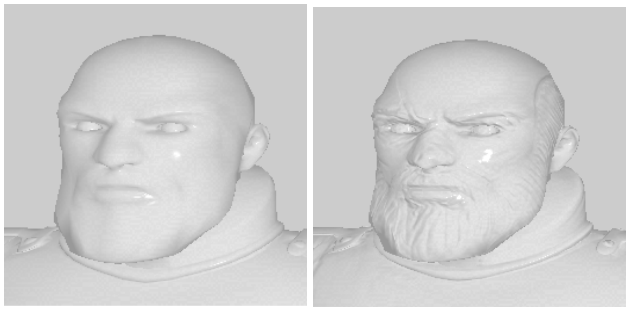


Figure 3. The part of low poly character model lit by Phong lighting model on the left and then the same model lit by Phong model with use of normal-mapping, which is rendered with more details (shows bumpy surfaces around character's eyes, nose etc.).

by itself does not produce realistic shadows (example in the picture 4), it only produces hard shadows and suffers from artifacts (more in section 4).

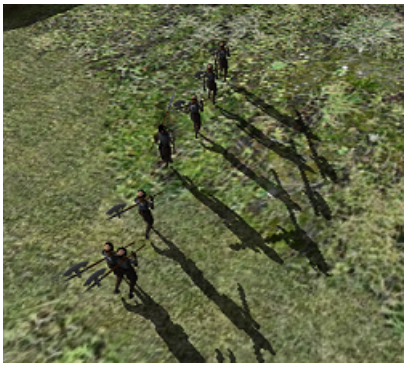


Figure 4. The group of characters casting shadows produced by basic implementation of shadow-mapping.

3.3 Deferred rendering

This technique [7] provides significantly better performance when rendering a scene with many lights than forward rendering and we can make use of it even when rendering outdoor scene affected only by one directional light. That is mainly because of the fact that during the process of this technique we create and fill so called G-buffer with geometry data (figure 5), which lighting or other (for example postprocessing) effects can profit from.

3.4 Screen space ambient occlusion

Screen space ambient occlusion (SSAO) [8] is approximation of how each vertex of scene is exposed to ambient lighting. This technique is suitable for real-time rendering because it is implemented without CPU overhead inside of fragment shader program and therefore it is completely executed on the GPU side. Our implementation samples each pixel with 4x4 texture

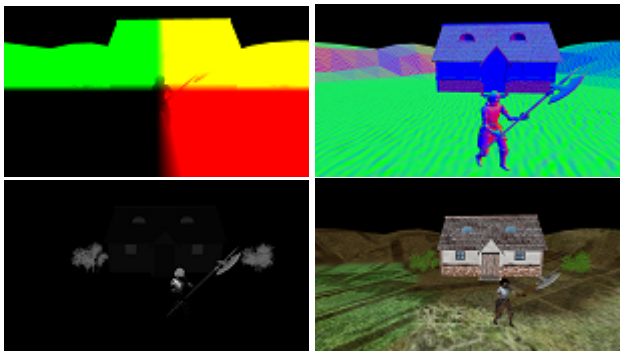


Figure 5. The content of G-buffer (information saved in set of textures) used in our application: **top left:** vertex positions, **top right:** normals, **bottom left:** specular color with shininess in alpha channel, **bottom right:** diffuse color.

reads (texture holding position of sampled pixels) and then calculates occlusion factor based on position of these surrounding pixels. This technique offers more realistic depth look of models (figure 6) at a cost of six percent frame rate drop[2] which is negligible for our purposes.

Figure 6. A model of a house rendered without SSAO (**top**) and the same house rendered with SSAO (**bottom**), where we can see added depth mainly around windows and door.

3.5 Skybox

Skybox [9] creates illusion of distant 3D surroundings of scene, makes scene look bigger than it is. Rendering distant buildings, mountains, etc. using geometry would be inefficient. Instead we can project these surroundings in a texture mapped on simple cube or other geometry like sphere/hemisphere (skydome). Figure 7 shows the power of skydome. In our application we use skydome instead of skybox, it requires more geometry to render but on the other hand it provides more flexible way of creating textures for skydomes (a panoramic picture can be transformed using polar coordinates into texture for skydome).

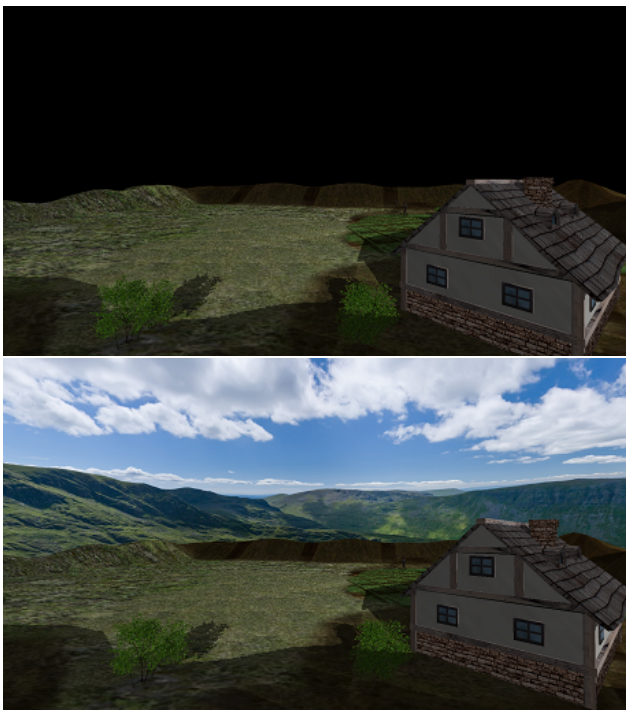


Figure 7. The scene rendered without skydome (**top**) and the same scene rendered with skydome (**bottom**), which makes the scene look bigger and more realistic with cost of rendering only extra 128 vertices.

3.6 Lens flare

Implementing lens flare [10] into our engine adds more dynamic range look of the scene. We simulated light scattering in lens system by postprocessing final image of rendering pipeline by sampling the brightest pixels in the centre of current frame along the vector from that pixel to the center of screen. This is multipass process in which we first generate lens features, after that we multiply the result with radial blur kernel and finally we blur the result.

4. Conclusions

This article presents methods and techniques used in developped application for character animations, autonomous behaviors of these characters and rendering technique to make scene look more realistic including advanced lighting effects and postprocessing effects

[2] 4×4 sampling kernel, resolution 1920×1080, NVidiA GeForce 840M

Figure 8. This figure shows the final application, rendered scene with all implemented effects and implemented GUI.

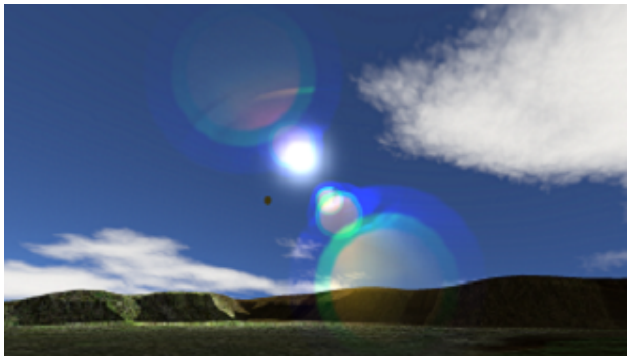


Figure 9. The result of postprocessing lens flare implementation.

(figure 9 shows implemented application at run time). We accomplished our goal to provide users with GUI for customizing visual appearance of rendered scene, customizing group behaviors and paths of walking pedestrians and posibility to record animated scene. Implemented GUI allows to import models (ASSIMP library supported file formats) to create custom scenes as well as import character models with their animations. Characters can be assigned to different groups and for each group properties like start, goal of the path, behaviors of characters in that group (characters keeping distance between each other, avoid collisions, etc.) can be customized. GUI also provides an easy way of customizing textures and visualization of meshes which create imported models. Another part of implemented GUI is customization of methods like lens flare effect, screen space abient occlusion, etc.

In conclusion implemented application is able to generate video sequences of easily customized scenes with walking pedestrians. It is able to render simulated character movement in real-time (up to 40 characters in more than 25 frames per second), thanks to generating videos frame by frame even really low frames per seconds in bigger scenes is not an issue.

In the future work I would like to implement more shader effects (bloom effect, antialiasing, etc.) along with optimalizations including space partitioning, level of detail, etc.

References

[1] Jing chao Chen. Dijkstra's shortest path algorithm, 2003.

[2] Bryan Stout. *Game Programming Gems*, chapter The Basics of A* for Path Planning. Charles River Media, Inc., 2000.

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

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

[5] Yanni Hajioannou. Gamedev glossary: What is a "normal map"? *Game Development*, 2013.

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

[7] Brent Owens. Forward rendering vs. deferred rendering. *Game Development*, 2013.

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

[9] Etay Meiri. Tutorial 25: Skybox. *OGLdev - Modern OpenGL Tutorials*, 2011.

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