Visionaire: Integration of 3D Characters into a 2D Adventure Engine

Daniel Cornel
0726194
Vienna University of Technology

Abstract

Although techniques of computer graphics have evolved a lot during the last years, there are still genres of computer games that rely on 2D graphics, especially point-and-click adventures. Still, 3D elements can significantly improve appearance and development of 2D adventures. This work focuses on the integration of 3D characters into 2D environments as well as techniques of computer graphics that are of particular importance for point-and-click adventures.

Common methods are discussed and evaluated with the aid of an implementation of 3D character support in the 2D adventure game editor Visionaire Studio [Visionaire Studio 2010]. This includes the import and rendering of animated Collada files, simple shadows, and toon shading, with a strong focus on usability and performance.

Keywords: 2.5D, adventure, video game, cel shading, toon shading, point-and-click

1 Introduction

An adventure game is a game that takes the player on a journey to foreign times and places to experience adventures, just like reading an exciting novel. Hence a good and compelling story is inevitable for the success of an adventure game. Furthermore, the player will always have to solve puzzles and interact with the game environment to push the story forward. In short, the main characteristics of an adventure game are narrative, puzzles, and exploration [Bronstring 2002], not graphics. A survey presented by Ju and Wagner [1997] revealed that the overall story is the main aspect of an adventure game to be fun, while the visual display is not a main concern. This is why many adventure games can still rely on 2D graphics and benefit from its advantages, especially point-and-click adventures.

As the term states, all or most of the interaction in point-and-click adventures is done with the mouse and therefore limited. In contrast to the movement in open worlds of modern 3D games, point-and-click adventures only allow very limited movement between predefined waypoints in predefined accessible areas. Interactions with objects or other characters in the scene are even more constrained. There is only a finite number of possible interactions in a point-and-click adventure which have all been scripted. The gameplay is usually time-independent and for a single player only. This linear structure suits well for storytelling and provides several simplifications in the development and display of such games for the sacrifice of action elements. The worlds of modern 3D games always contain regions of less importance where the player will not reside very long, but still, these regions have to be created and filled with vegetation and props or they will look sterile. In point-and-click adventures, almost every scene is significant for the story while unimportant locations are skipped. This helps to stay focused on the story and reduce the development effort at the same time.

In point-and-click adventures, most often hand-drawn or pre-rendered images are used for backgrounds and objects. They provide high quality because they can be produced with elaborate techniques such as global illumination and anti-aliasing, but can be displayed very performant. Yet, a disadvantage of pre-rendered images is the lack of motion which has to be faked using a sequence of images. To increase flexibility of animation, animated 3D objects can be used instead of sprite animation, which is still much more efficient than rendering the whole scene in 3D. However, the seamless combination of 3D objects and a 2D environment to so called 2.5D games takes some effort. To make a 3D object look like a part of the 2D scenery, it has to be colored and shaded as the 2D objects as well as being positioned between them correctly in terms of overlapping and shadowing.

The challenge is not only to achieve the seamless composition of all the various elements, but also to pick out the most suitable techniques for doing so. Similar to other computer games, the development of 2.5D adventures is bound to constraints like budget, performance, and backwards compatibility with old hardware. It is not useful to implement highly complicated algorithms into a game if there are simpler methods that lead to a similar result. This is why this document as well as the implementation focus on rather basic methods, like cel shading and blob shadows, that produce fair results within a short development time.
1.1 A word on 2.5D

In general, the term 2.5D outlines several kinds of dimensionality between 2D and 3D and is not unambiguously defined. It describes elements that are neither pure 2D nor 3D elements, i.e. the third dimension - usually the depth - is somehow limited. In video games, 2.5D also describes the interaction of 2D and 3D elements. This can be very beneficial because the advantages of 2D and 3D can be combined as long as it suits the gameplay. 2.5D video games can be roughly subdivided into the following three implementations.

1.1.1 Pure 2D with a faked third dimension

Appearance of 2D games can be significantly improved by adding the illusion of depth. Some of these techniques are adapted from classic painting, e.g. perspective projection and realistic lighting. Figure 2 shows the use of these two basic techniques in the 2D point-and-click adventure The Secret of Monkey Island. The whole game consists of 2D sprites that are composed on a plane. The perspective distortion that can be seen at the stairs and the railing leads to the illusion that the left part of the scene is closer to the camera than the right part. By scaling the characters along the y-axis they appear smaller in the background, i.e. the upper part of the scene, and bigger in the foreground to support the illusion of perspective. Additionally, the lighting of mast and canon leads to the impression of curved surfaces, while they are in fact plane.

Figure 2: Perspective projection and lighting in The Secret of Monkey Island.

A more advanced technique is parallax scrolling that uses several layers on which the sprites are arranged. A hierarchical composition of the layers allows sprites in the foreground to overlap sprites in the background. If the camera moves, sprites in the foreground move faster than sprites in the background to simulate the nearness to the camera [Jackson and Hague 2005]. This is a commonly used technique in side-scrolling games as well as every other 2D game that features a moving camera. Figure 3 shows a schematic representation of more than 20 different layers used for parallax scrolling in The Whispered World, using a modified version of the Visionaire Engine (see Section 2.1).

Sometimes even early first-person shooters like Doom or Wolfenstein 3D are considered 2.5D games not because of their display but because of their actually two-dimensional world and gameplay. The maps of Wolfenstein 3D are two-dimensional and do not allow height values at all. Doom maps allow one single height value per coordinate. That way, simple stairs are possible, but things like two-story buildings are not. Collision detection and aiming work in 2D, making it impossible to move the mouse up or down when aiming. Also, these games use 2D sprites instead of 3D models to represent enemies. They are rotated using billboarding to always face the camera [Patnode 2001].

1.1.2 Pure 3D with limited movement in the third dimension

Three-dimensionality is a quality label for video games, no matter if the third dimension really improves gameplay [Rouse 1998]. All point-and-click adventures could be done in genre-typical 2D without a negative impact on the gameplay, because gameplay itself does not usually profit from the additional dimension. Still, there are good reasons for creating an adventure game in 3D and then adopting 2D characteristics like a fixed camera afterwards. Instead of faking depth and perspective distortions, a game can easily calculate realistic values if it is completely three-dimensional. Lighting and shadow calculations can be done with existing algorithms instead of using own methods to simulate realistic 3D lighting and shadowing of 2D sprites. Adopting existing techniques and saving development time might be the main reason for producing games with two-dimensional gameplay in 3D.

In 2005, the point-and-click adventure Ankh, which uses the open-source 3D rendering engine OGRE with a fixed camera, was released. This way, the scene looks like any other 2.5D adventure, but uses the advantages of modern 3D graphics. When in a dialog, the camera shows the talking character head-on (Figure 4), occasional camera panning helps to stay focused on the character, and high quality video sequences can be done in-game very easy. Yet, the advantage of free camera movement like in other 3D games is
lost and although created three-dimensional, the world can only be
explorated from a single viewpoint. This limitation leads to the
classification as 2.5D.

1.1.3 A combination of 2D and 3D objects

When talking about a modern 2.5D adventure, it is most likely an
adventure that combines pre-rendered 2D sprites and 3D objects.
While all static objects can be 2D objects, all animated objects can
be animated in 3D. Thus, a reduction of development time and an
increase of rendering performance can be achieved at the same
time. Developers are neither bound to the creation of complex 3D
worlds when using 3D animation nor to the usage of 2D sprite animation
when using 2D sprite representations for some scene objects. As
this approach relies on 2D sprites, most enhancements of pure 2D
games can be used for 2.5D games as well, though with minor
changes. When combining objects with and without depth infor-
mation, decisions have to be made how these can collide, intersect,
and overlap.

This approach will be implemented into Visionaire Studio. There
are two major methods in combining 2D and 3D objects that will
be discussed in detail in Section 3.1.

Whenever there is a reference to 2.5D in the following, this defini-
tion is meant.

1.2 The evolution of adventure games

Is the development of point-and-click adventures still profitable? It
is important for publishers and developers to know if a game pays
off or if archaic game concepts are not in demand anymore. The
genre of adventure games is indeed over the hill, but nonetheless
there are genre fans who remake old adventures and create new
ones on a noncommercial base. There are even editors like Vision-
aire Studio that vastly simplify the development of point-and-click
adventures so everybody can create one, even without programming
skills. These people are not only driven by nostalgia, but by the
wish to tell a story. As already stated, the linear structure of 2D
adventures in general is ideally suited for storytelling.

In the late 1970s, the first adventure games were developed and,
since output of graphics was very limited, they were completely
text-based. To interact in the game Colossal Cave Adventure, also
known as ADVENT (see Figure 1 for an overview), for example,
there was a textual description of the scene and the player had to
type in every command by hand [McMurray 2010]. Compared
to today’s games, it was very inconvenient that input and output
were completely text-based, yet ADVENT already implemented the
three key characteristics of adventure games mentioned before.
Developers began working on graphical output and in 1984, King’s
Quest was released by Sierra On-Line. It established the usage
of third person characters that could be moved across the screen,
though it still used text-based input. It was not until 1988 that Lu-
casfilm Games developed the SCUMM system (Figure 2), a point-
and-click interface for 2D adventures, using a mouse as input de-
vice. Some of the most famous adventure games used the SCUMM
system, including The Secret of Monkey Island and Day of the Ten-
tacle, which are considered two of the best adventure games of all
time [Dickens 2004].

This time is considered the golden age of adventure games, not
least because of the rat race between Sierra On-Line and Lucas-
film Games. As adventure games became commercially successful,
a lot of competitors wanted a share of the pie, launching own games
and focusing on different aspects of the gameplay to be unique. The
downfall began with the revolutionary game Myst in 1993 [Roschin
2001]. Myst strongly focused on exploration and featured a first
person perspective, whereby every possible viewing angle was pre-
rendered in 3D. Additionally, details of the scene were animated
using video overlays. Thanks to the newly introduced CD-ROM,
this huge amount of data could be used. It was the best selling
video game until The Sims surpassed it in 2002 [McMurray 2010].
The commercial success of Myst was a signal to other development
studios to focus on graphics as well. Just like today, graphics be-
came a main aspect of most adventure games, often leading to a
lack of original concepts and neglecting the story and fun aspect.

The major developers Sierra On-Line and Lucasfilm Games tried to
shift to 3D games, but failed [McMurray 2010]. Sierra stopped pro-
ducing adventure games in 1999, Lucasfilm Games continued the
Monkey Island and Indiana Jones series with 3D sequels that did
not pay [McMurray 2010]. Instead, the new genre of action adven-
tures was born. Action adventures are a modern way of storytelling
in video games, although pure 3D graphics and action elements are
vital for the gameplay. The probably best known action adventure
Tomb Raider, released in 1996, is a good example for the slow but
steady disappearance of former concepts. First, action adventure
games use three or even four dimensions - if you count the time de-
pendence - that lead to a completely different gameplay [Roschin
2001], and second, the vast progress in computer graphics allows
games to look more and more realistic, turning their back on hu-
morous and comic-like style. There are some 3D point-and-click
adventures like Ankh or the later Sam & Max sequels that try to im-
ituate the comic look of former times. But in general, real-time 3D
graphics are not the best choice for a comic look with flat objects.

Figure 5: Screenshot of The Longest Journey.

However, some developers realized that pure 3D was not suitable
for their needs, went back to 2D and used 3D only to facilitate de-
velopment when appropriate [Boyes 2007]. Since then, especially
smaller European game development studios have been producing
classic point-and-click games.

The first notable 2.5D adventure was The Longest Journey launched
by Funcom in 1999, which had a very compelling story and yet -
thanks to the use of 3D characters and pre-rendered backgrounds
- very smooth graphics. It was followed by Microid’s games Sybe-
ria (2002), Syberia II (2004), and Paradise (2006), with the stories
written by the Belgian comic artist Benoit Sokal.

Notable are also the 2.5D point-and-click adventures Runaway and
its two sequels launched by the Spanish developer Pendulo Studios
in 2001, 2006 and 2009. These games display the 3D models with
toon shading and contours (Figure 6), giving them a great comic
look that will serve as an example for the implementation of toon
shading in Visionaire Studio (see Section 3.2).

Another example that will be analyzed in the following is the state-
of-the-art 2.5D adventure Secret Files 2: Puritas Cordis from 2009,
the sequel of Secret Files: Tunguska, developed by the German
studio Fusionsphere Systems.
As already mentioned, the German developer Daedalic Entertainment used a modified version of the Visionaire Engine for the award-winning 2D point-and-click adventures *The Whispered World* (2009) and *A New Beginning* (2010, Figure 7). Relying on hand-drawn images, these games show that point-and-click adventures are still in demand and that it still can be beneficial to prefer 2D to 3D graphics in order to tell a story.

2 Visionaire Studio

The basis of this paper is the implementation of 2.5D support into Visionaire Studio, which is done in a concurrent practical work. In Section 3, different possible algorithms are discussed for partial aspects of the 2.5D support, whereby only one is selected for implementation at each time. To understand the criteria for this selection, the current state of Visionaire Studio including its possibilities and limitations is presented in the following. In addition, the objectives and priorities of the implementation are explained.

2.1 Overview

Visionaire Studio is a free multi-platform software package that contains a WYSIWYG editor for the creation of 2D point-and-click adventure games using the *Visionaire Adventure Game Engine* as well as a player to run those games. There are different licensing models that allow both free distribution of games running on the supplied player and commercial distribution of standalone games, making Visionaire Studio suitable for all kinds of adventure game developers. The engine uses the OpenGL API for hardware acceleration and features common graphic effects such as particle effects and parallax scrolling. Further, the advanced resource management allows resources to be preloaded and freed automatically what makes the games very performant. The Visionaire Editor offers an own scripting system with support of the Lua scripting language, utilities for speech synchronization and animation, and a wide range of supported multimedia file types, so no other development tools or even programming skills are needed to create a game.

As the game only supports 2D graphics, the whole system is based on sprites. There is no third dimension describing the depth of objects, although an overlapping order and a rough guess of the center of objects along the z-axis, the object center, is defined by the game developer. Figure 8 shows how a horizontal line is drawn over the sprite of the large rock a little right of the center. This information is needed for the decision whether static sprites overlap with a moving character or not.

Like in other 2D adventures, Visionaire offers the simulation of a perspective projection by scaling movable objects along the y-axis. Since scaling of objects depends on their distance to the eye point which is not given in 2D, scaling factors have to be set by the developer as well. When the accessible areas and the waypoints for possible paths of a scene are defined (see Figure 9), the scaling factor for each waypoint is defined as well and will then be interpolated along the path at runtime. The scale factors can be a rather rough estimation because there are usually no situations where the game character has to walk deep into the scene or towards the camera. Hence, there should not be much scaling at all.

The game characters are represented by sprites as well, whereas animation is realized through sprite sequences. In the hierarchical data structure, a character has various outfits (see Figure 10 for the hierarchy in the editor). An outfit is a set of animations that are subdivided into walk, talk, standing, random and character animations. Each of these groups contains an animation for every direction that is needed in the game, e.g. walk up and walk down right. Finally, an animation contains a sequence of sprites. As a sprite animation requires quite a few frames to run smoothly, it is an enormous task for artists to create all necessary sprites. The demo game included in the Visionaire bundle has only one character and yet 568 different animation sprites. Loading all these sprites takes its time and when they have to be stored in the video memory uncompressed...
along with other character and scene sprites, video cards are driven to their limit.

2.2 Objectives

Primary objective of the 2.5D support of Visionaire Studio is the reduction of loading times and video memory requirements at runtime by avoiding sprite animations. This can also decrease development time spent on content creation. It is necessary that 3D models can be produced, textured, and animated in commonly used 3D modeling software such as Autodesk Maya or Blender and then be imported into the Visionaire Editor without modification. Since there are a lot of 3D file formats used in different applications and not each one can be supported, an interchange file format has to be used. The royalty-free XML schema COLLADA (.dae) suits perfectly for this task because all common 3D modeling tools support the export to a COLLADA file. Loading of the 3D files will be done using the Open Asset Import Library (Assimp) [Assimp 2010] which is an Open Source library that fully supports COLLADA. Assimp is released under the terms of the 3-clause BSD license and can thereby be used even with the commercial license of Visionaire Studio.

It is important to implement 2.5D support without affecting the existing code base and internal structures. The implementation is an extension to the 2D functionality, not a replacement, so no complete redesign of display functions or data structures can be made, although this limits the features that can be implemented. To remain consistent with the rest of the OpenGL-related code and to guarantee best hardware compatibility, only OpenGL features up to version 2.1 should be used for rendering. Furthermore, the whole implementation should be expandable in terms of additional algorithms for shadows and toon shading.

The user of the Visionaire Editor may have no knowledge about 3D rendering and shall not be confronted with implementation details. To ensure this, abstractions have to be made in the setup of the 3D space so that only a minimum of parameters have to be set by the user, e.g. the camera location and angle. There have to be meaningful default values and methods that are easy to use to change them in the user interface, of course. The aim is that 3D characters can be used in a game just as easy as 2D characters.

The best way to integrate 3D models into the data structure described in Section 2.1 is to extend an outfit with a new model data type. This way, 2D and 3D characters can be treated the same way in the game logic. The only distinction has to be made when displaying an animation. Since the sprite animations consist of image sequences, each animation is bound to the direction that the depicted character faces. Hence eight different sets of sprites are necessary for every animation to cover all possible directions an in-game character can face (0 to 360° in 45° steps). This overhead is not given a 3D character because it can be rotated to every direction before drawing, so only one 3D animation has to be stored in the outfit’s list of animations for an animation type. Instead of selecting a set of sprites to specify an animation in the editor, the user will have to load the 3D model and select one of the contained model animations. At runtime, the 3D model and its textures are preloaded when necessary and deleted when not required any longer.

As the look of a game depends on its graphics, the display of the 3D characters has to be adjustable in order to make them match with the rest of the scene. This requires realistic lighting and at least simple shadows for the use of realistic looking pre-rendered backgrounds as well as toon shaded rendering for the use of hand-drawn or otherwise cartoon-like backgrounds. This adjustment has to be done by the user, so again it is inevitable to provide methods that are easy to use to change the settings in the user interface. Currently, shadows have to be drawn onto the sprite (see Figure 10) to be displayed in-game. This is a very inconvenient way that will be simplified with the automatic calculation of shadows (see Section 3.3.3).

3 Techniques

3.1 Realization of 2.5D

As Visionaire already supports parallax scrolling and the simulation of perspective projection, this section focuses on the implementation of the 2.5D definition described in Section 1.1.3. There are basically two different ways of combining 2D spaces and 3D spaces when the camera is fixed.

3.1.1 Using a single 3D space for all objects

The obvious and usual way is to overlay the 2D content with an invisible 3D space (Figure 11). It is set up by defining a ground object where the characters can walk, limited by borders in order to make it impossible to walk off the scene. These definitions can be quite difficult to set and require 3D-related knowledge as well as three-dimensional imagination.

The 2D scene shows some kind of space where the 3D objects shall be integrated into, so 3D information has to be extracted from the background in order to adjust the 3D space [Petrović et al. 2000]. In Appendix A, a possible method for manual extraction is sketched. It rotates the 3D scene and a visible ground object to match with perspective characteristics like the horizon of the 2D scene. The rotation angle then can be used to set up the camera of the 3D space. A similar method was described by Petrović et al. [2000]. This is known as camera calibration because the camera rendering the 3D space is aligned to match the eye point and orientation that was used when creating the background image. Automatic camera calibration is a well-researched topic, as techniques for 3D reconstruction from 2D images and videos depend on the right orientation [Liebowitz 2001]. Fortunately, camera calibration is only necessary if the information cannot be retrieved in another way, i.e. when the scene is hand-drawn or a photography. As many 2.5D point-and-click adventures use pre-rendered backgrounds, the original camera settings can be adopted from the rendering software. Yet, the necessary extraction of 3D information is a huge disadvantage for games with hand-drawn backgrounds which forces developers to deal with 3D-related problems.

Another issue to deal with using a single 3D space for all objects is...
(a) Front view of the pre-rendered background (semi-transparent). Dummy objects (blue) are placed in the scene. There is a single point light (green, in the upper right) for all objects.

(b) Perspective view of the scene. The camera (red) is the viewpoint of (a). All dummies have the same size and stand on the ground (gray).

**Figure 11:** Setup of a single 3D space for all 3D objects. The 2D background will be displayed using an orthographic projection, the 3D scene behind the background will be displayed using perspective projection. That is why 3D objects in the back appear smaller. (a) shows a schematic front view with dummies arbitrarily placed in the 3D space behind the background including different z-values. In a game, the dummies are drawn on the background.
the missing connection between the 2D scene and the 3D scene. To get both of them together at runtime, there are three options: (i) The whole 3D scene is drawn over the 2D scene. This means that sprites of the 2D scene cannot overlap 3D characters. (ii) The whole 2D scene is drawn over the 3D scene. Although this was done in Figure 11(a) using a semi-transparent sprite to show the 3D characters, this is obviously not suitable for real games as most of the time all 3D models would be hidden. (iii) The 2D sprites are drawn in the 3D space. This is the common way because all objects are visible and can be overlapped, but it requires some effort. Since 2D and 3D scene are not connected in any way, overlapping calculations cannot be done using regular depth-testing. Instead, the depth information of the objects has to be made comparable.

However, there are some more advantages of using a raw 3D scene in the background, such as realistic lighting calculations. Figure 12(b) shows the content of an Autodesk 3ds file that is attached to the Wintermute 3D characters technology demo. The light sources (white) are not only drawn onto the 2D background, but also placed in the additional 3D scene and are used to illuminate the 3D characters. This also allows shadowing on scene objects, which is very useful for games in general, but rather unimportant for 2.5D adventures (see Section 3.3). In Figure 12(a), it is clearly visible that the shadow of the character can be cast onto the stack of boxes, which is represented by a 2D sprite. This is only possible because the boxes are also included in the 3ds file.

The use of a single 3D space for all 3D objects requires camera calibration or 3D information about the 2D elements, advanced depth sorting algorithms, and about all a lot of work with 3D-related topics for a non-3D game, so where is the benefit? It is the perspective view of all models. A single 3D space with a single viewpoint allows the imitation of natural-looking perspective distortion. This makes graphics more realistic on a very subtle way. Characters are displayed with foreshortening effects and the distant characters appear smaller than characters closer to the view plane [Hearn and Baker 2004]. Although the camera is static in most 2.5D adventures and the scenes are rather small, perspective distortion is visible, especially when a character walks across the screen. An example is given in Figure 13. With the character staying at the leftmost reachable point of the scene (Figure 13(a)), the character is visible angular from the front, whereas it is visible angular from behind when staying at the rightmost reachable point (Figure 13(b)). The pose is exactly the same both times.

Yet, there is a slight disadvantage with perspective projection as well since it only works for the 3D objects, not the static scene. Usually, this is not a problem at all because characters are the only significantly moving objects of a scene, but when using a side scrolling effect for larger scenes, everything is moving for a short amount of time. The common implementation of side scrolling tries to display a walking character in the middle of the screen if possible, for the player has the best overview over the scene then. When a walking character passes the middle of the screen and the scene is scrollable, the camera is moving along the x-axis with the character until the player stops or leaves the scrollable area which is given as

\[
\frac{\text{width}_{\text{screen}}}{2} < x < \frac{\text{width}_{\text{background}} - \text{width}_{\text{screen}}}{2},
\]

assuming that \(x = 0\) at the left margin of the scene.

Side scrolling is used to display scenes that are wider than the current screen resolution which is the case when the scene was pre-rendered using a wider horizontal point of view (Figure 13(c)) than the camera that is rendering the 3D scene at runtime and moving it (Figure 13(d)). While the camera is moving along the x-axis (indicated by the gray arrow in Figure 13(d)), the focal point of the

Figure 12: Support of complete 3D scenes in the Wintermute Engine. Note that a pre-rendered background is displayed in (a).
Figure 13: Perspective distortion and scrolling artifacts in Secret Files 2: Puritas Cordis.
Camera is moved as well so that there occur no changes to the perspective distortions of the moving character while scrolling. However, the movement of the camera leads to missing distortions of the 2D scene objects which are expected because their distance and position to the camera change. For example, the chair in the middle of Figure 13(c) is displayed from the side. When the camera shows the left part of the scene (Figure 13(a)), the backrest should be clearly visible. This is not the case because the field of view of the 2D scene is not changed, but the pre-rendered image is cropped to fit the screen.

This is just a minor inconsistency, but it might be more disturbing for some players to have inconsistent foreshortening than none at all. There are several ways to deal with this problem:

1. It can simply be ignored because it is not vital for gameplay and does not occur very often at all.
2. It can be concealed using parallax scrolling. While scrolling, some parts of the scene will move faster than others, leading to the impression of perspective distortion. This method was used in the discussed scene by placing furniture closer to the camera at the bottom of the screen (see Figures 13(a) and 13(b)). This does not fix the problem, but it is a good compromise.
3. Side scrolling and otherwise moving camera effects can be completely avoided.
4. It can be fixed by using 3D geometry for the static scene. A 3D replica of the scene can be imported, textured, and then used instead of the 2D background, but this actually leads to a pure 3D game. A more appropriate way is to use camera projection (also called camera mapping).

Camera projection is camera based texture mapping, i.e. the texture is projected onto a 3D surface from the perspective of a fixed projection camera [Bowden 2005]. This actually combines the colors of the texture with the depth values of the 3D shapes and allows the former 2D scene to be viewed from different angles to a certain extent. The 3D shapes do not have to be detailed since the details are already included in the texture, so again an additional low quality 3D scene file can be included in the game. This method is perfectly suitable for 2.5D adventures because there is very limited camera movement and therefore no risk that a player could see the unprojected rear of the 3D scene.

A proof of concept of camera projection by Crytek employee Hannes Appell [2010] is shown in Figure 14, using original 2D backgrounds and concept art of the Moniker Island 2: LeChuck’s Revenge to rebuild various scenes in the CryEngine. As the texture is distorted when being projected on the 3D surfaces (Figure 14(c)), a post-processing step is necessary to polish the look of the scene.

In general, realistic foreshortening effects, lighting and shading are not needed for 2.5D adventures, but can improve appearance and game atmosphere. There are several ways to overcome the difficulties with camera calibration, depth sorting, and side scrolling, especially when using an auxiliary 3D scene. This turns the use of a single 3D space for all 3D objects into the common procedure for 2.5D realization and has been widely used in commercial games since the beginning of 2.5D adventures. Yet, it is too extensive for the use in Visionaire Studio because it requires three-dimensional imagination and advanced knowledge of 3D-related processes to use it right in the editor what would exclude some users.
(a) Front view of the scene. Dummy objects (blue) are placed in an own 3D space with an own camera (red) and point light (green). Each camera renders its object into a texture that is used by a sprite and attached to the background.

(b) Perspective view of the scene. All dummy sprites lay on a XY plane using different Y values and scaling factors.

**Figure 16:** Setup of a scene using an own 3D space for each 3D object. An object is rendered to a texture that is placed on the screen as a sprite. All sprites will be displayed in the game using an orthographic projection. Note that the dummy positions of (a) and Figure 11(a) almost look the same.
3.1.2 Using an own 3D space for each object

Another and more user friendly way to implement 2.5D is to use an own 3D space for each 3D character, including an own camera and light source. The idea is to map the 3D geometry onto a plane before using it in the game (see Figures 15 and 16). That way, it can be rendered into a texture, applied to a 2D sprite, and then be used in the 2D scene like every other sprite. This is very useful if an already existing 2D engine like Visionaire Studio shall be extended. For the user this is easier to handle because the only 3D-related step that has to be done is to import the character. The cameras do not have to be calibrated and the models do not have to be set in the 3D space. Additionally, it does not require sophisticated depth sorting algorithms and basically circumvents all disadvantages of a single 3D space. But the same also holds for the main advantages.

Movement of characters is done by moving their corresponding sprites on the 2D scene and scaling them. While moving across the 2D scene, the model is rotated into the walking direction and one of the predefined 3D animations of the model file is played, although the model’s location in its own 3D space always stays the same. There are a lot of parameters that can be set to adjust the camera settings, but as said in Section 2.2, to keep it simple the system is abstracted where possible, so the user will only have to set the camera tilt angle $\beta$ (see Figure 17), the camera distance and the camera height once per scene. The character’s viewing direction $\gamma$ is calculated in a previous path finding step.

As every character is handled independently from the others and from the eventual location in the scene, the scene context is lost. Without a global 3D space where 3D characters can interact with each other, no shadows can be cast from one character onto another. When every light source has to be defined locally and its position is static in the isolated 3D space of its character, characters cannot be illuminated by global light sources visible in the scene. For the characters are rendered head-on by their cameras before being placed in the scene at runtime, the foreshortening effect has to be waived as well.

On the other hand, this also eliminates the inconsistency of some objects being distorted correctly, while others are not. In some cases, the foreshortening is not even desired, especially when a flat cartoon-like look shall be achieved.

The result of this approach looks and behaves like sprite animation, but without its limited use of different viewing angles and, of course, the huge amount of data. For the development of 2D adventures, there are tools like the 3D Ripper DX that load a 3D model, allow the user to set static light sources and the viewing angle, play the included animations and then create sprite sequences from it.

(a) The tilt angle is limited to $-90^\circ \leq \beta \leq 90^\circ$ because the other angles can be achieved by rotating the character (see (b)) by $180^\circ$.

(b) The character can only be rotated around the y-axis by the angle $\gamma$.

The same functionality is included into a game, only that rendering happens at runtime and to a texture that is used immediately. This is very useful for the implementation in Visionaire Studio because the main objective is to substitute the expensive creation and loading of sprite animations, but not their easy handling at runtime. Additionally, shadows can be calculated and stored in the texture as well, using a texture with alpha channel for partial transparency.

The implementation in Visionaire Studio uses textures of fixed size and optional MIP maps for all characters in order to render them at maximum resolution. This is important because each character is scaled down on loading depending on its maximum extents to ensure that every part of it is inside the view volume of the camera, no matter how it is rotated. Nevertheless, it has to be drawn without scaling artifacts, so it is rendered at the defined maximum resolution - 1024 * 1024 pixels for example - and MIP maps for smaller sprites can be created. On later video cards even multisampling can be used within framebuffers to avoid aliasing artifacts, resulting in very high quality of the output texture. This is of major importance, especially when integrating the characters into hand-drawn
or cartoon-like scenes because artifacts convey the impression of trashy technology used for the game and might distract the player. Since graphics are only the tool of traditional adventure gameplay, they should be unobtrusive and supportive, not disturbing.

Although using an own 3D space for each object perfectly suits the needs of Visionaire Studio, it is not used in commercial games because professional game developers have the skills and required 3D data to implement the approach of 3.1.1.

3.2 Toon rendering

![Figure 18: A teapot rendered by a photo-realistic algorithm (left) and by a toon rendering algorithm (right).](image)

Toon rendering is a non-photorealistic rendering technique that is used to imitate the look of pen and ink drawings or comic style drawings [Danner and Winklhofer 2007]. There are various applications for toon rendering because of its inherent abstraction which makes it a very suitable tool for visual understanding. Technical manuals for example most often involve simplified and toon rendered illustrations rather than photographs [Gooc et al. 1998].

As the term states, toon rendering has its origins in the style of cartoon movies and comic books with characteristic contours and solid colors (see Figure 18). This technique is not surviving because it is easier than photorealism, but because the focus on stylistic and artistic aspects are often more important. A less photorealistic look helps media to communicate thoughts, emotions, and feelings as well as it enables an audience to immerse in the story told. It leaves space for imagination if a character is not realistically portrayed but only sketched and people find it easier to identify with the character. Thereby toon rendering is an attractive technique for video games that gives reason to expect a more compelling storytelling experience [Lake et al. 2000].

Until now, toon shading is sparsely applied in modern video games because the trend still goes to photorealistic rendering, but there are also exceptions such as the 2.5D adventure game series Runaway (see Figure 6) and the multiplayer 3D game Team Fortress 2 by Valve. While Runaway utilizes toon rendering to look like traditional hand-drawn point-and-click adventures, it is a novelty in a 3D action game like Team Fortress 2 and was used not only to create a unique look in contrast to the various photorealistic action games, but also help the player with visual aids [Mitchell et al. 2007].

Basic toon rendering will be used in the Visionaire Engine out of consideration for games with hand-drawn images because it enables the 3D characters to look like hand-drawn, too.

Cel shading is often used synonymous to toon rendering although technically speaking, it only describes the shading methods and not the silhouettes. The origin of the term is celluloid, the early medium that traditional cartoon animation was drawn onto. Usually the foreground figures of an animation frame were outlined on celluloid and later colorized, before being recorded on a background [Isenberg 2008]. As the silhouette edge detection (SED) and shading are likewise different tasks and research topics, they are discussed separately in the following.

3.2.1 Silhouettes

![Figure 19: Different types of silhouettes: Silhouette edges (green), ridge edges (red), valley edges (yellow), boundary edges (blue), texture edges (cyan). The last picture shows a complete silhouette image. Occluded edges are not visible.](image)

Silhouettes are viewpoint dependent lines that mark discontinuities in the projection of an object and can be subdivided into four different types (see Figure 19) [Kang and Kim 2001; Wang et al. 2004]:

1. **Silhouette edges** are edges with one adjacent face being a front face and one being a back face. Having a solid object, these include the outlines as well as internal discontinuities like the hole in a donut, for example. Let \( V \) be the viewing vector and \( n_1 \) and \( n_2 \) the normal vectors of the two adjacent faces, then a silhouette edge is detected if

   \[
   (n_1 \cdot V) \times (n_2 \cdot V) \leq 0. \tag{2}
   \]

   The term is also used to generally describe the difference to edges which are not detected, so a silhouette edge might be this specific edge type or just one of the four presented. For better understanding, the term is only used for this specific edge type in this document, whereas contour edge is used to describe all edges that should be detected by a SED in general.

2. **Crease edges** are edges between two front faces with the dihedral angle deceeding a given threshold \( t \),

   \[
   n_1 \cdot n_2 \leq t. \tag{3}
   \]

   **Ridge edges** are the crease edges that point out of the model, **valley edges** are the crease edges that point into the model.

3. **Boundary edges** are edges with only one adjacent face and should not be present in solid (closed) objects.

4. **Texture edges** are no real edges but discontinuities of the texture mapped on an object. It might be desirable for some applications to generate silhouettes for the texture as well, e.g. if the belt of a game character shall be accentuated but is only present in the texture and not in the geometry to reduce the polygon count. Another reason for the consideration of texture edges in the SED is to generate silhouettes for billboards...
such as distant trees to make them appear as complex 3D objects.

Note that the characterization of an edge does not mean it is visible as it could also be occluded.

Silhouette edges can be drawn with minimum effort with the use of back faces. The back faces are drawn in wire frame mode with thick lines in the desired color before the front faces are drawn in regular mode. The back faces will be occluded by the front faces, but not the outermost lines due to the thickness of the lines. While very efficient, this is not always useful because crease edges are neglected [Gustafsson 2008]. Therefore, sophisticated detection algorithms are used. Most SED techniques focus on silhouette and crease edges because they are the most expressive features of the object. Most SED algorithms work in either image space or object space, but it is preferable to use techniques that operate on local data for the use in a shader.

**Image-space** algorithms detect silhouettes using image processing algorithms on rendered intermediate results in image buffers and therefore do not need any 3D information. Usually different edges are detected with different methods and then combined to a result, as demonstrated by Decaudin [1996]. The diagram of this algorithm is shown in Figure 20.

Most silhouette, boundary, and texture edges can be detected in an image rendered into the color buffer without any illumination that is then checked for discontinuities convolving it with an operator for edge detection such as the Sobel operator [Hertzmann 1999]. The disadvantage of this method is that silhouette edges between two objects of identical color are not detected, whereas texture edges are also detected, even if this is not wanted [Wang et al. 2004]. To avoid dispensable edges, the detection can be performed in the depth buffer. Instead of the color values, the depth values of an intermediate image are used. Again, only silhouette and boundary edges can be detected and edges between two different objects next to each other will not be detected if their depth values do not differ significantly.

If crease edges are to be detected as well, an intermediate normal buffer that contains the components of the normal vector for each pixel can be used. As the normal vector of a face corresponds to the direction it is facing, normal vectors of the faces adjacent to a crease edge differ notably. The normal buffer can be set up by rendering the scene with the help of a shader to an intermediate RGB texture. The per-vertex normals are interpolated for each pixel when being handed over from the vertex shader to the pixel shader and can be stored in the RGB texture as the pixel color [Decaudin 1996; Wang et al. 2004]. Once the edges are detected and stored in edge images, they can be combined with the results of the shading steps.

The complexity of SED in the image space is independent from the geometry, but proportional to the image resolution [Danner and Winklhofer 2007]. The described methods can be implemented using modern graphics hardware for buffer creation and visibility calculations. The combination of depth buffer and normal buffer produces satisfying results. They are therefore very efficient and scale well, but as they operate in the image space, contour edges are represented by pixels. All 3D geometry information is lost what makes it difficult to use them in later rendering steps, e.g. styled edges could be occluded [Wang et al. 2004].

**Object-space** algorithms avoid these problems because first, they use and preserve the object information for the calculations, and second, detection and visibility calculations are usually separated. Thus, visible styled contour edges can be drawn after the front faces to avoid unwanted occlusion. The obvious way of detecting the edges is to create an edge list with adjacency information, loop through all edges, calculate the dot products needed for the categorization of the edge, and then mark it. This method is guaranteed to detect every silhouette of the types 1 to 3, if desired. Yet it is a brute force approach that performs bad on complex polygons because the calculations for every edge have to be repeated every frame. Furthermore, it does not consider coherency of edges within one frame and between frames [Gustafsson 2008]. A probabilistic SED algorithm was presented by Markosian et al. [1997]. With the observations that (i) most edges are no contour edges, (ii) a contour edge always has an adjacent contour edge and (iii) with no or minor changes to the camera position and orientation, the silhouettes are similar or equal to those of the last frame, it is possible to reduce the necessary calculations drastically. All edges are sorted by their probability to be a contour edge, whereas edges with higher dihedral angles are more likely to be contour edges than others. Additionally, contour edges of the last frame are more likely to be contour edges in the current frame as well. According to this probabilities, only a small amount of promising edges has to be checked which is done in a random way. If a checked edge is a contour edge, all adjacent edges are checked as well. This way, all contour edges connected with the first one will be detected. This algorithm is very fast, but there is no guarantee that all contour edges are detected, although the amount of edges to be checked can be raised and it will still be more efficient than the brute force approach.

Beside these two algorithms, there are several other approaches such as using pre-computed edge information. However, as this requires the user to create the edge information, it is not suitable for use in Visionaire Studio and is not discussed here. A detailed overview over SED algorithms was given by Gustafsson [2008].

![Figure 21: Use of stylized contour edges.](image-url)

Another interesting aspect of object-space algorithms is the stylization based on object information. Silhouettes can be drawn with strokes stored in textures instead of using just straight lines. By calculating the angle of an edge with its successor edge for example, it can be decided if the silhouette shall be drawn with a straight or a curved stroke, leading to smoother outlines (see Figure 21). This is of special interest for video games because smooth curved surfaces are often desired, but not realizable because of the huge amount of polygons necessary. It is therefore very handy if the edged look of polygon models can be attenuated. Instead of solid black strokes, every other line type or texture can be used as well, giving the opportunity to imitate every favored drawing style [Lake et al. 2000].

In summary, object-space algorithms neglect the potential of graphics hardware because most of the calculations have to be performed on the CPU. On the other hand, they extend the possibilities for stylization. It is not said that object-space algorithms are better or worse than image-space algorithms, but that the decision is to be made whether spare CPU or GPU resources should be used for the SED and whether stylized contours should be displayed.
Figure 20: Diagram of the toon rendering algorithm presented by Decaudin [1996] as an example of image-space SED.
The third stage to perform the SED is in the geometry shader, where the mesh is available at primitive level with adjacency information and the possibility to create a new geometry in the output. With the given adjacency information, silhouette edges including crease edges can be detected like in the object-space algorithms, but much faster than on the CPU. A polygon can then be created out of a detected edge and passed to the pixel shader. When creating texture coordinates for the edge polygons, it is even possible to texture them to achieve strokes and other stylizations [Hermosilla and Vazquez 2009]. Unfortunately, the geometry shader is a relatively new feature that was first introduced with Shader Model 4.0 and is only supported in OpenGL 3.2. As one objective of the implementation is to guarantee best hardware compatibility, this approach cannot be used in the Visionaire Engine.

Adventure games usually do not use the CPU or the GPU to capacity and the complexity of the geometry is manageable. This is why it is planned to implement an object-space algorithm for the SED in the Visionaire Engine at a later stage of development in favor of possible stylizations such as strokes. In the meantime, the very efficient approach of drawing thicker back faces is used which produces very promising results already.

3.2.2 Shading and highlights

![Figure 22: Two- or three-dimensional impression depending on shading. Left to right: Diffuse shading, no illumination, toon shading.](image)

The aim of toon shading is to let 3D surfaces look flat and colorized by hand. As noticeable in Figure 22, the three-dimensional impression of an object depends on the illumination. Traditional cartoons as well as some hand-drawn games such as The Curse of Monkey Island renounce illumination, resulting in a flat and maybe cheap look. It is therefore beneficial to vary the tones on a surface slightly to distinguish between the lit and shadowed sides of an object [Johnston 2002]. The compromise between completely flat and realistically lit objects is to use a toon shading method. The intent of toon shading is to use a well-known diffuse illumination model for calculations, but display only a few different tones to indicate the illumination. As specular highlights are usually neglected in games, the shading is view-independent and cannot represent metal or other materials where highlights are of importance [Barla et al. 2006].

Using a single light source, the illumination $I$ of a surface point is given as

$$I = I_a k_a + I_d k_d \max(0, n \cdot l)$$

where $I_a$ and $I_d$ are the intensities of the ambient and diffuse light, $k_a$ and $k_d$ are the ambient and diffuse reflection coefficients at the point, $n$ is the surface normal vector at the point and $l$ is the unit vector in the direction of the light source [Gooch et al. 1998].

![Figure 23: 16 texels wide 1D texture with three intensity values.](image)

Instead of using the clamped Lambertian term $n \cdot l$ directly, it can be mapped onto a 1D texture which contains regions of constant intensity values, as shown in Figure 23. As texture coordinates are usually given as values between 0 and 1, $\max(0, n \cdot l)$ can be used to look up the intensity to use in the 1D texture. It is very simple and yet very effective, as seen in Figure 22.

![Figure 24: 2D texture containing different levels of detail.](image)

There are several modifications of basic toon shading, most often using another mapping function. Barla et al. [2006] demonstrated the use of a 2D texture, where the second dimension is the aimed-at level of detail of the solid color bands (see Figure 24). Involving view-dependent terms in the calculation, objects closer to the camera can be drawn using hard transitions between two tones, whereas distant objects can be drawn blurry using a lower level of detail, for example.

![Figure 25: 1D texture used in Team Fortress 2.](image)

In the video game Team Fortress 2, the discrete regions of constant intensity are substituted by the 1D texture shown in Figure 25. It contains the whole continuous transition from light to dark, but it is tightened and beyond the transition, constant intensities are used. The result also contains the two major tone areas, light and dark, but with smooth transitions which make the whole model look toon-like and still not hand-drawn (see Figure 26). Additionally, Phong and rim lighting are included in the pixel shader and no silhouettes are drawn at all, so the toon rendering methods used are not intended to imitate a hand-drawn look, but a unique and aesthetic one. As stated by Mitchell et al. [2007], the primary goal of the shading was to get a clean and flawless look that does not distract the player. It suits very well for fun games including action games and has
been adapted by several other games such as the free-to-play game *Battlefield Heroes*.

In Visionaire Studio, the support of 1D textures is sufficient because information like different levels of detail is rather unimportant in a scene with a fixed camera. To achieve both hard and soft transitions between the intensity bands, it is possible to generate a texture for a given number of different intensities and to import an own texture to be used.

Until now, only the use of diffuse illumination has been discussed, since most traditional cartoon movies and comic books did and do not usually use specular highlights. They can, however, help depicting a special material or curvature of a surface. The small visual hints do not have to be realistic as long as they help with the perception of a scene, such as adding black or white parallel lines to a transparent surface to indicate a window [Winnemöller and Bangay 2002]. To display specular highlights, a specular term such as the Phong term simply has to be added to Equation 4, leading to

\[
I = I_a k_a + I_d (k_d \max(0, n \cdot l) + k_s (R \cdot V)^n s) 
\]

where \( k_s \) is the specular reflection coefficient at the point, \( n_s \) the shininess of the surface, \( R \) the unit vector in the direction of ideal specular reflection, and \( V \) the viewing vector [Hearn and Baker 2004].

Figure 27: *Toon shading with regular (left) and more aesthetic specular lights (right).*

However, realistic highlights often do not look very convincing in cartoon scenes. Adopting the style of Japanese manga and anime with large and sometimes disconnected highlights, scenes can look more appealing (see Figure 27). Anjyo and Hiramitsu [2003] presented an algorithm to generate this kind of highlights. With a modified version of the Blinn shading model, a set of operations involving the halfway vector is defined which can be understood as modifications of the highlight itself. In this way, highlights can be translated, stretched, rotated, split, and squared until they look like the result in Figure 27.

This is a very useful effect for games that try to copy the style of manga, but as with specular highlights in general, the use of stylized highlights is neither required nor exceedingly useful when imitating the look of traditional comics for games. Especially for games with a fixed camera such as 2.5D point-and-click adventures, the amount of calculations necessary for the specular illumination is much more of a drawback than the gameplay could benefit from the view-dependent highlights.

### 3.3 Shadows

Having implemented 2.5D support and toon rendering in Visionaire Studio, the games can already look very good - depending on the artist of the models and textures. The techniques presented so far are no magic bullet, but at least the artificial look of 3D models is reduced. Still, the comprehensibility of a scene can be further improved with shadows. Not to make the scene look more realistic - although this can be achieved as well -, but to give the player a clue of the non-existent 3D space of the whole scene after the composition with the 3D characters. Shadows are crucial for the human perception of three-dimensionality, for they help clarifying ambiguities in the display of objects [Hasenfratz et al. 2003].

Figure 28: *Information provided by shadows.*

Shadows often provide information about the position, size, and geometry of the occluder as well as the geometry and position of the receiver (see Figure 28). Additionally, they can provide information about objects currently not visible, for instance the shadows of birds on the ground.

Since shadows have become commonplace and almost indispensable in real-time 3D applications, a wide range of different algorithms exists to realize them, each of them having its advantages and disadvantages. This is not the place to discuss them all, as this has elaborately been done by Hasenfratz et al. [2003] and Eisemann et al. [2009] recently, to name only two. Instead, a short overview of the major different approaches is given that can be subdivided into hard shadows and soft shadows. Additionally, a method for faked shadows called blob shadows is presented in the following.

#### 3.3.1 Hard shadows

Hard shadows are produced by point light sources, as shown in Figure 29, where only the decision is made if the light source is visible from the surface point or not, leading to hard transitions between the *umbra* and the lit part of the scene. A point light is assumed because it simplifies the calculation, although it does not exist in...
practice and it will only look proper if the distance from the light to the
occluder is much larger than the distance from the occluder to the
receiver. Therefore hard shadows in any application are always
a trade-off between realism and efficiency. However, they are the
most common shadows in modern computer games because they
can be computed with reasonable effort. The two basic techniques
are shadow mapping and shadow volumes [Hasenfratz et al. 2003].

The idea of shadow mapping is to determine if a surface is visi-
ble from the point of view of the light source. Every hidden element
lies in shadow. To determine visibility, the scene is first rendered
from the light’s position into a depth buffer, whereby for each pixel
the z-value of the first visible surface is stored. This buffer is called
the shadow map. The creation of depth maps is very efficient due to
graphics hardware support because it is the same mechanism used
to resolve visibility during standard rendering.

In a second pass, the scene is rendered from the actual viewpoint.
For each rasterized fragment of the scene, its position is trans-
formed into light space. With the transformed x- and y-values, a
lookup can be performed in the shadow map. The stored depth
value is then compared to the transformed z-value of the fragment.
If it is larger than the distance stored in the shadow map, it lies in
shadow, otherwise it will be lit [Eisemann et al. 2009]. This way,
self-shadowing is also handled.

As the algorithm requires a viewing direction of the light source, it
is actually an algorithm to be used for directional lights only. To use
it for omni-directional lights, the current standard procedure is to
use six shadow maps that correspond to the faces of a cube around
the light source. This way, all directions can be captured, but the
whole scene has to be rendered six times instead of once to get the
necessary information. There are, of course, other algorithms that
try to overcome this disadvantage, such as Dual-Paraboloid Shadow
Mapping, which only requires the scene to be rendered twice for the
shadow maps [Eisemann et al. 2009].

Shadow mapping can produce very satisfying results, but due to
its discretization, the basic algorithm suffers from significant alias-
ing artifacts. When the shadow map is rendered, the scene is reg-
ularly sampled from the perspective view of the light, leading to
oversampling of near objects and undersampling of distant objects.
When using a single shadow map for a large scene, thin objects
might not be sampled and therefore do not cast a shadow, and stair-
stepping artifacts can occur to undersampled shadows. Addition-
ally, imprecision of the compared z-values leads to shadow artifacts
on lit surfaces. In sum, the discretization in shadow mapping is a
major disadvantage that leads to incorrect results. The error can
be minimized by increasing the precision like using huge shadow
map resolutions, but this is not always feasible because of hardware
constraints. Since shadow mapping is a reliable algorithm, many
modifications have been made to correct the shortcomings, such as

Adaptive Shadow Maps and Tiled Shadow Maps which use hier-
archical data structures to refine the shadow map [Eisemann et al.
2009].

At a subsequent stage of development, shadow mapping will be in-
tegrated into Visionaire Studio because it can be used efficiently in
the presented implementation of 2.5D (see Section 3.1.2). The arti-
facts caused by an undersized shadow map can be avoided because
all possible positions of the characters are known, allowing the de-
veloper to adjust the size of the shadow map without undersizing or
unnecessary oversizing.

A similar method is applied in a lot of 2.5D games using a single
3D space for all characters, computing all shadows separately with
a local light source (see Figure 30). The geometry is projected on
the receiver, but no self-shadowing is done. This way, the shadows
are not realistic because they may not correspond to the shadows
of the pre-rendered scene or the positions of global light sources,
but this will hardly be noticed. Computing shadows this way is
very convenient because the individual shadow maps do not have to
cover the whole scene and can be much smaller. Additionally, pro-
jecting the shadow straight to the bottom circumvents the problem
that shadows cannot be cast on objects of the 2D scene realisti-
cally without underlaying 3D geometry. Note that in Figure 30, the
character is illuminated from the right, but the shadow is projected
downward. Also, the translucency of the large pre-rendered shadow
on the ground and the shadow of the character differ and the projec-
tion of the character shadow can darken the already shaded region.
These are, however, only minor flaws of 2.5D games which do not
affect the gameplay.

The shadow volumes algorithm basically finds the silhouette of an
occluder along the light direction, extrudes its edges along the light
direction into infinity and thus creates a shadow volume. Surface
points inside the shadow volume are in shadow, surface points out-
side are lit [Hasenfratz et al. 2003]. In theory, the determination if
the surface point is inside a shadow volume can be done by casting
a ray from outside the volume to the surface point and counting the
intersections with the volume borders. In practice, the amount of
intersections that have to be calculated every frame is very high. To
avoid this problem, methods based on the stencil buffer are used.
One of these is the Z-pass method which shoots the ray from the
view point. First, the depth buffer is filled by rendering the scene
with ambient light. Then, the stencil operation is set to increment
and the front faces of the shadow volumes are rendered with en-
abled depth test. After that, the back faces are rendered, decre-
menting the stencil buffer. Shadow volume fragments further than

![Figure 29: Geometry of hard shadows.](image)

![Figure 30: Use of individually computed shadows in Secret Files 2: Puritas Cordis.](image)
the surface point will not pass the z-test, leaving the stencil buffer entry set to 1. In the last step, the scene is rendered with the desired shading model where the stencil buffer is 0. The major advantage of this approach is that no geometric intersections have to be used. But Z-pass will produce wrong results if the viewpoint itself lies in shadow.

The Z-fail technique tries to overcome this by counting the shadow surfaces behind a surface point instead of those in front of it. First, the depth buffer is filled again, then the stencil operation is set to increment on depth fail. This way, only shadow surfaces behind the surface point are counted. Then, the back faces of the shadow volumes are rendered. In the following step, the stencil operation is set to decrement and the front faces of the shadow volumes are rendered, leading to the same results in the stencil buffer as those of the Z-pass method. For this technique, the shadow volumes must be capped [Hasenfratz et al. 2003; Eisemann et al. 2009].

Shadow volumes have some advantages over shadow mapping, such as the inherent support of omni-directional light sources and eye-view pixel precision shadows. Yet, they could not be used in real-time applications for a long time because of the high amount of calculations necessary. With techniques such as Z-fail as well as with vastly increased performance of graphics hardware, it is possible today to use shadow volumes in real-time applications [Eisemann et al. 2009], as demonstrated with the implementation of shadow volumes in Doom 3. But it is obvious that this algorithm is not appropriate for 2.5D games as its results cannot top those of shadow mapping for a scene with manually sized maps.

### 3.3.2 Soft shadows

In reality, light sources have an expanse and do not produce hard shadows. Instead, there are points on the surface of the receiver where only a part of the light source is visible, therefore it is lit with less intense. The continuous transition from the umbra - if existing - to completely lit regions is called penumbra and makes the shadow smooth. Although realism is not a criterion for the implementation in adventure games, a smooth look is. Soft shadows do look less artificial in games than shadows with stair-stepping artifacts. As the exact computation of the penumbra would require a large number of calculations, real-time soft shadows are generally generated by using hard shadow techniques and create the penumbra from the umbra.

As complex shadow computations are not profitable for scenes of manageable size, there is only given a short overview over how shadow mapping can be extended to a smoother look with minimum effort. The advantage of soft shadow algorithms extending shadow mapping is that it can be adopted with manageable effort at a later time into an engine, if needed.

The simplest method is to blur the shadows in image space. It is, however, not useful to blur the shadow map itself, for it only contains the depth values, so blurring them would decrease the precision of the map and lead to even more artifacts. Instead, the shadows can be rendered into a screen-sized buffer from the viewpoint of the camera. The result can then be blurred in one or multiple passes using a common filter, such as Gaussian, and will be projected onto the scene using screen-space coordinates. Depending on the intensity of the filter, this will generate smooth transitions between umbra and lit surfaces, although they are not physically correct. It may happen with thin objects that the shadow is blurred beyond the object boundaries, leading to a floating penumbra.

As the shadow map depends on the position of the light source, it is also possible to render it several times and move the light source slightly each time. This will produce a slightly different shadow map every time, whereby an area light source can be approximated. The different shadow maps can be averaged in an attenuation map which produces very good results. Yet, for the computation of the attenuation map the scene has to be rendered several times, making it very inefficient [Hasenfratz et al. 2003].

Another approach is to use more than just one depth value to determine whether a surface lies in shadow or not. Percentage-Closer Filtering for example tests, for each point of the scene, depth samples from a constant window around the projection into the shadow map. The results of these tests are weighted and averaged, by what the binary decisions of shadow maps are exceeded. Other approaches include Variance Shadow Maps and Convolution Shadow Maps. However, Percentage-Closer Filtering already produces fair results at reasonable effort and is therefore quite popular in video games [Eisemann et al. 2009].

### 3.3.3 Blob shadows

Blob shadows are unrealistic shadows that are almost independent from the light source as well as from the geometry of occluder and receiver. As demonstrated in Figure 32(b), it is not necessary to
project the accurate geometry of the occluder to identify its shadow. Instead, a texture of the desired shadow (see Figure 32(a)) can be mapped onto a plane positioned on the ground below the occluder. If the ground is not plane, the y-value of the shadow plane has to be evaluated each frame, since there is no projection step. The quality of blob shadows is therefore dependent from the scene, so they are unfavorable in scenes with bumpy ground because the shadow plane may intersect with it. But, when using it in 2.5D games, the ground of the 3D space will be plane most of the time.

On the other hand, the only information of the occluder’s geometry needed to create such a shadow is an estimation of its diameter to initially scale the shadow plane accordingly. At runtime, the plane will be scaled and faded proportionally to the the height of the occluder. The initial translucency can be either set manually or controlled by the intensity of the light source. Blob shadows are not a real shadowing technique but a hack that tries to offer the benefits of shadows without costly calculations. They are therefore often used in video games as a fallback shadowing technique if more realistic shadowing is too expensive. In Visionaire Studio, blob shadows are implemented as the first shadowing technique and will be complemented with shadow mapping later.

4 Conclusion

The main differences between 3D and 2.5D games as well as the reason why 2.5D still works in contemporary video games have been stated. The main concern of adventures are narrative, puzzles, and exploration, so the game engine has to be built on these requirements. In contrast to modern 3D games in general and first-person shooters in particular, the adventure genre is not a playground where new graphic effects should be tested or introduced. Graphics are not the engine of an adventure, but an expedient for the representation. Therefore, development of adventure games is guided by unobtrusive techniques rather than realism and treats for the eyes.

When developing 2.5D games, it is important to specify which kind of look is to be achieved and what techniques might be of help. When being integrated into hand-drawn scenes, 3D characters look best when shaded flat and with silhouettes, whereas for pre-rendered scenes, Phong shading with soft shadows might be preferable. To achieve a humorous look, toon shading with several bands of color and hard shadows can be used.

All these types of adventures will be supported in Visionaire Studio, with no constraints regarding the combination of different features. With the specific implementation of 2.5-dimensionality with a single 3D space for each character, a maximum compatibility with the existing code and usage can be achieved for the good of the users. Along with this, no made decisions limit the capabilities of the Visionaire Engine and the presented methods to be implemented are basic and lightweight algorithms that can be extended any time. Thus, all objectives defined in Section 2.2 are achieved with the presented extensions to the existing code, while the good usability is preserved.

5 Acknowledgments

Thanks to Martin Mayer of Fusionsphere Systems for his information about the implementation of Secret Files: Tunguska and Secret Files 2: Purtias Cordis. Also thanks to Johannes Letzgus for his help with Figure 13 and to Tabea Cornel for proofreading this work.

List of Figures

9. Camera projection used to rebuild a scene of Monkey Island 2: LeChuck’s Revenge in the CryEngine. Retrieved from http://www.youtube.com/watch?v=PPaH8o1D77g
10. A teapot rendered by a photo-realistic algorithm and by a toon rendering algorithm. Retrieved from [Decaudin 1996]
14. 2D texture containing different levels of detail. Retrieved from [Barla et al. 2006]
15. 1D texture used in Team Fortress 2. Retrieved from [Mitchell et al. 2007]
17. Toon shading with regular and more aesthetic specular lights. Retrieved from [Anjiyo and Hiramitsu 2003]
18. Information provided by shadows. Retrieved from [Hasenfratz et al. 2003]

References


A An example of manual camera calibration

Figure 33: The hand-drawn or pre-rendered background image.

When using a pre-rendered or hand-drawn background image (Figure 33), the camera that renders the 3D space in the game has to be adjusted so that 3D objects and the 2D scene can be composed without perspective error.

Figure 34: Side view of the two scenes in 3D.

Figure 34(a) shows an imaginary 3D scene as source of the 2D background, since it was drawn or rendered from a viewpoint \((0, h, 0)\) looking into the viewing direction. Here it is assumed that the viewing direction was only tilted by the angle of \(\alpha\), what makes it easier to explain the principle.

3D space and camera are set up as in Figure 34(b). The camera is set to \((0, \hat{h}, 0)\), while \(\hat{h}\) is a rough guess of \(h\). A visible ground object is used to define the walkable area. The 3D characters are placed where they are supposed to be in the game.

Figure 35: The whole scene is rotated by \(\hat{\alpha}\).

Looking from the viewpoint, the 3D scene is rotated around the x-axis (Figure 35) until the far end of the ground object is in line with the horizon of the 2D background as in Figure 36(a).

When the horizons shown in the Figures 33 and 36(a) match, the rotation angle \(\hat{\alpha}\) of the scene is a good approximation of \(\alpha\), the angle of the camera tilt in Figure 34(a). The scene is reset and the camera is rotated by \(\hat{\alpha}\), shown in Figure 36(b).

Figure 36: Generalized example of finding the right settings for the camera of the 3D space to integrate 3D characters into a 2D background.

The camera settings are now found and the 2D and 3D objects can be combined at runtime. If the 2D background image is displayed using an orthographic projection and then overlaid by the 3D scene using a perspective projection, the result will look similar to Figure 37. This approach is inaccurate, but as 2.5D adventures use static scenes, the 3D camera only needs to be adjusted so that no visible perspective errors occur.

Figure 37: The composed result at runtime.