The use of Imposters in Interactive 3D Graphics Systems

Kenneth Rohde Christiansen*

Department of Mathematics and Computing Science
Rijksuniversiteit Groningen
Blauwborgje 3
NL-9747 AC Groningen

k.r.christiansen@student.rug.nl / kenneth@gnu.org

Abstract
When developing an interactive 3D graphics system the developer needs to keep in mind that the system should be able to render more than 20 frames per second in order not to disturb the user. This unfortunately sets quite some limits on the complexity and amount of objects viewable by the user at a given time, and various methods have been developed in order to decrease the amount of work needed to render a given scene. One of these is *dynamic generated imposters*, which takes advantage of the similarities between each frame when the viewing position have only changed marginally. In this paper we will introduce the technique and look a bit more into detail on how to implement this using OpenGL. The document assumes that the reader has followed the Advanced Computer Graphics and Virtual Environment course given at the RUG† or at least a similar course given at another research institution.

Keywords: 3D Graphics and Realism, Real-time Rendering, Viewing Algorithms, Computational Geometry, Computer Science.

1 Introduction

In a virtual reality system the idea is to immerse the user into a computer generated world by presenting reality like (synthesized) data to the senses. When the frame-rate drops below 20 frames per second the user does not any longer perceive the individual frames as a continues motion. The user get dusturbed, with the result that interaction with the system becomes troublesome and the user might be unwilling to use the given system [Helman, Aug. 1993]. Given the fact that displaying realistic environments require hundred of thousand texture-mapped polygons, meeting a frame-rate above 20 becomes unrealistic using all but the most expensive hardware.

Due to the fact that movement in the 3D environment is often done smoothly (thus small changes in viewing and spatial position from one frame to the other) much work is spend re-rendering major parts of the geometrical database without the user being able to notice any difference. This especially counts for distant objects which often only consists of few pixels.

Having this in mind, the *dynamically generated imposters* idea [Schaufler, 1995] have been developed. The basic idea is to render some of the objects to texture buffers and instead of showing the actual object an *imposter* (ie. the texture) is displayed. Since this texture can be reused over a series of frames, many calculations are saved with the result that more complex scenes can be rendered at

*The actual proof-of-concept implementation have been developed in cooperation with my fellow student Martinus Bodewes - bodewes@fmf.nl

†Rijksuniversiteit Groningen, the Netherlands

the same frame-rate.

Many ideas have been proposed for limiting the calculations per frame and thus improving the frame-rate. Some of these will be discussed in section 2. Section 3 explains the basic idea of dynamically generated imposters and section 4 gets a bit more into the details needed in order to actual implement the idea. In particular, the important billboarding technique is explained. In section 5 and 6 we look at the results of our proof-of-concept implementation and concludes the presentation by summarizing the paper and mentioning a few senarioes where it might not be possible to use imposters.

2 Previous Work

Over the years many algorithms have been proposed in order to increase the frame-rate of these real-time 3D systems and many are also in use today, for instance in various computer games. You should keep in mind that many of these techniques actually can compliment the dynamically generated imposters technique.

2.1 Culling on non-visible objects

One of the most knows ways to reduce the amount of rendering work is to only render objects visible to the user. Finding the part of the geometry database can be hard, but many efficient algorithms have been developed based on a spatial organized geometry database [Clark, Oct. 1976]. The particular idea is used in the popular *Quake*[1] games series from id Software. When playing Quake you will notice that the game consists of many rooms and that doors often lead to small hallways and not directly into new rooms. This way the game seldom has to render objects from more than one room at the time and very simple and quick algorithms can be used for culling objects not visible to the player.

2.2 Levels of Detail

Another idea, also often used in 3D games, is using several levels of details, often refered to as LOD. LODs are thus hierarchical models of decreasing complexity. The idea is only to use the highly detailed models for close objects and use the lower detailed models for distant objects. There exists many ways to automatically generate these LODs [Heckberg and Garland, May 1994]. In [Rossignac and Borrel, Feb 1992; Schaufler and Strzlinger, Jan 1995] methods for creating these from polygonal patch models are discussed, in [DeHaemer and Zyda, 1991] how to generate from 3D digitized data and in [Turk, July 1992] how to automatically create from the tessellation (often *triangulation*) of curved surfaces.

2.3 Pre-generated imposters

Dynamically generated imposters are based on the idea of imposters, which is also refered to as billboards[2]. Standard imposters as described in [Maciel and Shirley, 1995; Nieder et al., 1993] are pre-generated and stored in the geometry database. This technique has a number of shortcomings that dynamically generated imposters tries to forecome. Since the imposters need to be pre-generated the developer needs to decide which viewing direction to store and at what texture resolution. High resolution and many pre-calculated viewing directions makes storing the textures in the geometry database impractically because of storage concerns. On the other hand, using low texture resolution and few viewing directions dramastically limits the usability of the imposter as they can only be used for certain viewing directions and distances.

3 Generating imposters dynamically

Imposters are transparent polygons (called billboards) with an opaque texture mapped onto them. The texture is generated at a particular view and re-used over a series of frames, as long as the viewing angle only changes marginally (lets say 20 degrees). In that case, the imposter texture will have to be re-generated.

The texture resolution needs not to extend the resolution of the screen, and can be closen lower for distant objects in order to save texture memory. If this is done, the imposter texture will also have to be updated when the distance to the object

[1] A popular first-action shooter from id Software. A demo can be downloaded from the company website, `http://www.idsoftware.com`

[2] In this paper, the term billboarding is used for describing a part of the technique behind imposters.

changes more than a given amount. A formula as the following can be used:

$$texture\ resolution = screen\ res. \times \frac{object\ size}{distance}$$

Generating the imposter is fairly cheap and it is hardly more costly than rendering the actual object onto the screen. A few extra steps are required, though, and even though that there are not that costly in software, most newer graphics cards usually provide hardware support (probably due to the use of imposters in popular video games).

The steps are:

- Clean the texture buffer.
- Setting up the view for rendering the object.
- Render a part of the view (the size of the objects bounding box) onto a texture that is stored in texture memory.
- Placing a billboard in the virtual world and rendering the imposter texture onto it.

4 Implementation

4.1 Choice of Platform

In order to get an idea of the usefulness of dynamically generated imposters and to see how easy it is to use, we implemented a proof-of-concept version with the use of the OpenGL[3] toolkit. It is a widely used 3D drawing toolkit of which there also exists an open source implementation called Mesa[4]. Most graphics card manifactures, such as NVidea, provide native drivers such that most of the functions are being performed directly in hardware.

Though, highly cross-platform compatible, OpenGL is loosing ground on the Windows platform due to Microsoft pushing their competitor *DirectX*[5]. The gaming industry is increasingly investing in DirectX as new versions are periodically released which takes advantage of the newest features of the graphic cards and because it makes it possible to easily port the games to the reasonable popular gaming station, the Microsoft XBox. Due to the fact that what pushes graphic card manifactures is the gaming industry, OpenGL drivers are today getting second-priority. This does not bring us from the fact that most virtual reality systems are based on OpenGL. During our course at the RUG we have, among others, worked with SGI Performer[6] and VRJuggler[7]; both tools based on OpenGL and used at the RUG for developing applications for the CAVE[8] With this in mind, we find OpenGL a good platform for testing the imposter idea.

4.2 Other implementation details

Since we have been developing a proof-of-concept implementation, no special details have been taken to performance and thus, no performance measurements have been taken. This is mostly due to the fact that before implementing imposters we had to implement a basic scene with basic navigation. In order to stress test the system we would need a semi-large geometry database, which we didn't have at hand. Searching the internet made us realize, that though many 3D objects can be downloaded for free, most of these come without textures which made them useless for our purpose. Additionally, most objects are distributed as 3D Studio Max meshes and writing an 3D Studio Max importer would bring us away from our original goal.

We ended up finding some GPL'ed[9] example code that could read a subset of the 3D Studio Max format as well as a spaceship object which we then used for our implementation.

A demonstration of the implementation can be seen in figure 1.

[3] More information about OpenGL can be found on the official OpenGL website, `http://www.opengl.org/`

[4] The open-source Mesa project is coordinated at the website, `http://www.mesa3d.org/`

[5] Many good resources about DirectX are available at `http://www.microsoft.com/windows/directx/`

[6] Please consult the official website for more information, `http://www.sgi.com/products/software/performer/`

[7] Please consult the official website for more information, `http://www.vrjuggler.org/`

[8] The CAVE (Cave Automated Virtual Environment) is a room used to project a 3-dimensional virtual reality illusion. The American organization NCSA defines the CAVE as "an immersive virtual reality facility designed for the exploration of and interaction with spatially engaging environments".

[9] GNU Public License: For more information please look at `http://www.gnu.org`

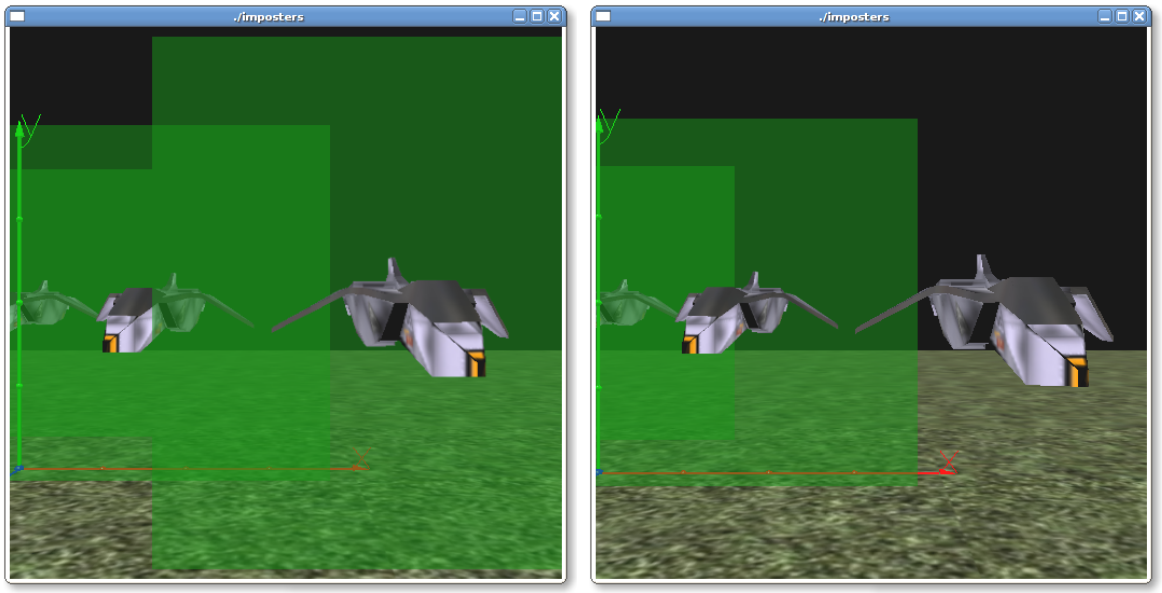

Figure 1: Shows the implemented prototype with imposters shown with a green tint. In the second image we moved one step closer and the real model is shown for one of the three spacecrafts. The difference is minimal and the user does not notice unless paying special attention.

Implementing billboarding is one of the major parts of our imposters implementation, so we will go into details with it in the following section. Basically, it is quite simple to implement when you have given it some thoughts.

4.3 Billboarding

Billboarding is a technique not limited to be used by imposters. Billboarding basically means adjusting the objects orientation so that it faces some target like for instance the camera, which is the case with imposters.

Billboarding allows us to replace a geometry figure with an imposted texture applied on a texture quad or two triangles. The billboarding guaranties that the texture is always facing the camera. In some environments some objects are used so often or are so complex that you never want to render them more than once. This often applies to such objects as grass and trees, and you will notice in various games that such objects are always pointing with the same side towards you. Since the objects are always pointing towards the user, the user handly realized that it is in fact just a flat texture quad. This illusion could be broken if the user could fly over the object.

For the use of these latter objects, spherical billboards are often used; billboards that, thus, rotate around the vertical axis. For the use of dynamically generated imposters, spherical billboards are more appropriate as they rotate around the horizontal axis as well. The latter is what we want to use for implementing imposters.

4.3.1 How to implement billboards

Billboards are easy implementable with OpenGL, as the `modelview` matrix stores all the geometrical transformations, rotations, scales and translations that you do. This matrix contains all the transformations required to change your input coordinates (world coordinates) into camera coordinates.

The matrix looks as follows:

$$\begin{bmatrix} a_{00} & a_{10} & a_{20} & a_{30} \\ a_{01} & a_{11} & a_{21} & a_{31} \\ a_{02} & a_{12} & a_{22} & a_{32} \\ a_{03} & a_{13} & a_{23} & a_{33} \end{bmatrix}$$

The top 3 values at the right-most column at the current position relative to the cameras position and orientation, and the top 3×3 submatrix beginning with a_{00} and ending with a_{22} contains scaling and orientation operations.

If we substitute this submatrix with the identify 3×3 matrix, we have in fact reversed the scaling and rotation operations and made sure that the billboard is always pointing toward a plane perpendicular to the camera. We have created a so-called *spherical* billboard.

$$\begin{bmatrix} 1 & 0 & 0 & a_{30} \\ 0 & 1 & 0 & a_{31} \\ 0 & 0 & 1 & a_{32} \\ a_{03} & a_{13} & a_{23} & a_{33} \end{bmatrix}$$

Since scaling is ignored we might have to scale afterward:

```
billboardBegin();
  glScalef(2,3,2);
  // draw imposter
billboardEnd();
```

The submatrix consists of 3 columns: the first is the *right* vector, the second is the *up* vector and the third one is the *lookAt* vector. In case we wanted to use the billboard for trees, gras as discussed earlier, we do no want the the billboard to be spherical but instead cylindrical. If this is what we want we do not touch up vector stay in the submatrix.

$$\begin{bmatrix} 1 & a_{10} & 0 & a_{30} \\ 0 & a_{11} & 0 & a_{31} \\ 0 & a_{12} & 1 & a_{32} \\ a_{03} & a_{13} & a_{23} & a_{33} \end{bmatrix}$$

What we have created here might be sufficient but we can do a bit better by making the billboards point directly toward the camera and not towards a plane perpendicular to it. In order to accomplish this we need to rotate the billboards manually.

For simplifying our calculations we assume that the *right vector* $= \{1, 0, 0\}$, *up vector* $= \{0, 1, 0\}$ and the *lookAt vector* $= \{0, 0, 1\}$. This means that we are drawing the object in the local origin looking long the positive z-axis. This is no problem as we can easily translate the billboard after we have rotated it correctly.

We want to rotate the object around the up vector by the angle between the vector from object to the camera and the *lookAt* vector. The object-to-camera vector can easily be found by substracting the position of the object from the position of the camera. The projection can also be found by setting the y component to zero. The angle (expresses as a function of cosine) can be determined by the inner product between the *lookAt* vector and the projected *object-to-camera* vector as long as *object-to-camera* vector is normalized. We will need to calculate the cross product between the vectors as well due to the fact that $\cos(\alpha) = \cos(-\alpha)$. In the case that the angle is positive, the cross product (a vector) will be pointing in the positive y direction and otherwise in the negative y direction. Rotating around this cross product vector we do the trick as if the vector points in the negative y direction he rotation will effectively be reversed.

This technique basically gives a cylindrical billboard. In order to create a spherical version a few extra steps are needed. Basically we need to tilt the object until it truely faces the camera. The additional rotation (tilting) has to be performed with the *right* vector as the axis. The cosine of the angle can be obtailed by the inner product between the *object-to-camera* vector and the **projected** *object-to-camera* vector.

This is the method that has been implemented in our proof-of-concept implementation.

4.4 Rending to offscreen buffer

When we want to render the object to an offscreen buffer and thereby create an imposter texture, we first need to positioning the object. By rendering the object below the ground floor of our scene we

make sure that only the object is seen by the camera. We position the object according to the orientation it would have in the real world and render it to an offscreen buffer with transparent background.

We are not describing the actual way to do this here in this paper as there are several. For instance, in the case you are using the NVidea driver it is possible to use the hardware accelerated `pbuffer`. We implemented a non-hardware accelerated method. The interested reader can find several examples on how to accomplish this on the Internet by using one of the search engines such as Google[10].

In our proof-of-concept implementation we worked with a fixed sized texture and billboard. In a real implementation the texture and billboard size should depend on the bounding box of the object, but due to us only working with one object implementing this did not seem valuable to implement.

4.5 Generating new imposters

The idea is not to generate new imposters for each new frame, but reusing the same imposter for a series of frames. As mentioned before, imposters give a very similar final image as long as the angle is not changed too much. In our implementation we decided to generate new imposters when the angle had changed more than 5% (18 degrees) in vertical or horizontal orientation. Because of this, we calculated the change in the horizontal and vertical orientation since the last time the imposter was updated and used these angles for positioning the original object before rendering it to the offscreen buffer.

4.6 Drawing order

Since the imposter textures are drawn with an transparent background, the order the billboards (with the textures drawn onto) are drawn is important as showin in figure 2. If we start by drawing the most distant billboard it will be taken into account when a partly transparent billboard is drawn in front of it. If not drawn in this order, you will not be able to see the distant billboard in the transparent parts of the covering billboard in front, which is obviously not what we want.

[10]The popular Google Internet search engine can be found at http://www.google.com

4.7 When to show imposters

The idea is to show imposters for distant objects. However since it should be basically be impossible to see the difference between imposters it might be an idea to let the test depend on the texture resolution chosen for the imposters and showing the imposter when a pixel on the imposter is equal or smaller than one pixel on the screen. This test can be performed by comparing the angle of the *field of view* to the screen resolution per pixel with the angle of the *field of view* to the maximum texture resolution per pixel [Schaufler, 1995]:

$$\frac{\alpha_{fov,\ texture}}{max.\ texture\ res.} < \frac{\alpha_{fov,\ screen}}{screen\ res.}$$

In our implementation we simple chose to show the imposter when the distance to the object from the camera exceted a fixed number. When showing the imposter we painted the texture with a green alpha tint which made it quite easy seeing when we were looking at the original model or the imposter.

5 Results

We have to admit that we were quite astonished with the result. It was practically impossible to see the difference between the imposter and the object directly drawn on the screen when we generated new imposters for each frame. When we reused the same imposter for a series of frames, it was only possible to notice that we were looking at an imposter because we knew that on forehand.

6 Conclusion and Future Works

In this paper we have looked at a method for speeding up the rendering of complex scenes in order to improve the frame-rate and thereby the user experience. The idea was to use so-called imposters and reuse these over a series of frames. The imposters were re-generated when the diverged too much from the original object due to the user/camera having turned more than a given degree.

The method is not all that new, and is for instance in use in some of the most popular first-person-shooters. In order to look at the usability of the method we did a proof-of-concept version

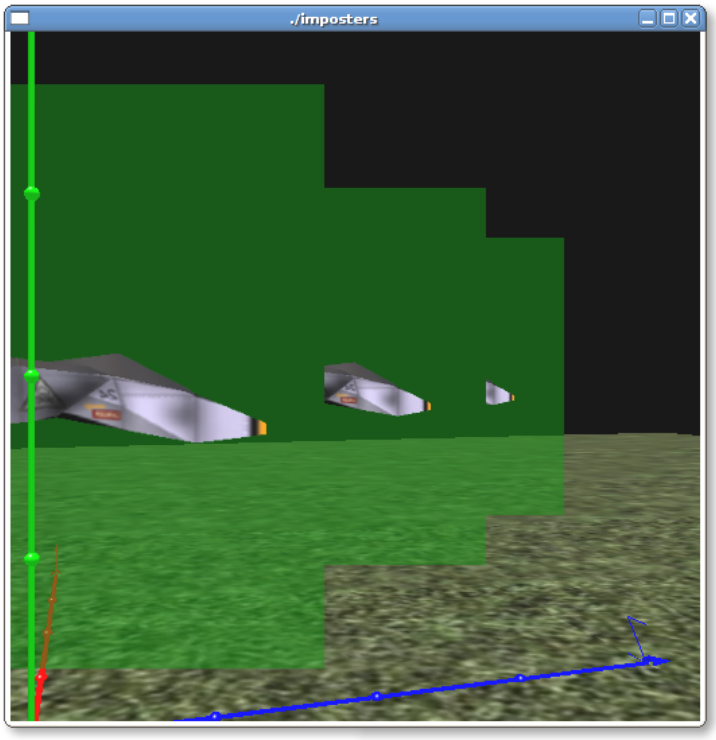

Figure 2: Shows the problem when not paying attention to the drawing order of the imposters. Since the imposter is drawn with transparents background, only what is currently on the image is taken into account, and thus not the imposters further away, which are drawn later.

and conclude that the negative visual impact is neglectable. Dynamic generated imposters is a easy-to-implement method with almost no visual impact that *with a good implementation* has the abbility to speed up the rendering of complex interactive 3D environments. It would be interesting doing actual measurements with a more elaborate implementation.

There are a few cases where it is not possible to use imposters. One of these is when we are dealing with intersecting objects. A semi-solution to this problem would be to consider the intersected object as one, and add them to the same imposter. Another case is when we are dealing with animated objects. Since presumable only few objects will be animated in the environment the technique will still improve the frame-rate.

7 Acknowledgements

We want to thank our teacher Jos Roerdink for an interesting and worthwhile course. We would additionally like to thank Martinus Bodewes for great discussion about the technique and for helping with the proof-of-concept implementation.

References

Clark, J. H. (Oct. 1976). Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, **19**(10), 547–554.

DeHaemer, M. J. J. and Zyda, M. J. (1991). Simplification of objects rendered by polygonal approximations. *Computers and Graphics*, **15**(2), 175–184.

Heckberg, P. and Garland, M. (May 1994). Multiresolution modeling for fast rendering. *Proceedings of Graphics Interface '94*, 43–50.

Helman, J. L. (Aug. 1993). Designing virtual reality systems to meet physio- and psychological requirements. *Applied Virtual Reality Course (ACM SIGGRAPH '93)*, 5–1 – 5–20.

Maciel, P. W. and Shirley, P. (1995). Visual navigation of large environments using textured clusters. *Interactive 3D Graphics 1995*.

Nieder, J., Davis, T., and Woo, M. (1993). *OpenGL Programming Guide*. Addison-Wesley Publishing Company and Silicon Graphics, ISBN 0-201-63274-8.

Rossignac, J. and Borrel, P. (Feb 1992). Multiresolution 3d approximations for rendering complex scenes. *IBM Research Report RC 19697 (#77951)*.

Schaufler, G. (1995). Dynamically generated imposters. *GUP, Johannes Kepler Universität, Linz, Austria*.

Schaufler, G. and Strzlinger, W. (Jan 1995). Generating multiple levels of detail for polygonal geometry models. *2nd Eurographics Workshop on Virtual Environments 95*, **ISSN 1024-0861**.

Turk, G. (July 1992). Re-tiling polygonal surfaces. *Computer Graphics (ACM SIGGRAPH '91 Proceedings*, **15**(4), 55–64.