5.7

Impostors: Adding Clutter

Tom Forsyth, Mucky Foot Productions

tomf@muckyfoot.com

Impostoring is a term that is probably not familiar to many reading this. However, the concept may be—it has surfaced in various forms many times in the history of 3D graphics. Simply put, it is about using sprites in a 3D scene, but instead of an artist drawing or rendering the sprites beforehand, they are updated on the fly. (In fact, the academic version of impostoring started when rendering things such as cityscapes for VR fly-throughs, and was not updated dynamically—that came in later versions. However, the dynamic update version is far more interesting to games, since by their nature they deal with scenes that change.)

Instead of rendering a high-triangle object every frame, the high-triangle object is occasionally rendered to a texture—usually on the order of once every 5 to 50 frames. Every frame this texture is mapped onto a much lower-triangle object which is drawn in place of the complex object.

The main target for impostors is scenes with lots of small static objects in them—clutter. Each of these objects will use an impostor, and most will be redrawn at a much lower rate than the main scene's frame rate. By doing this, the perceived triangle density is far higher than the actual density, and the visual quality allowed by all these incidental objects considerably increases the realism of the scene. The main difference between an office or house in a computer game and the real thing is the amount of clutter present.

Although newer cards have huge triangle throughput, using impostors is still of great benefit. The bus bandwidth (in the case of the PC, this is the AGP bus) is usually a bottleneck, and reducing this for some objects allows greater triangle detail to be used on others. An impostor is a single texture, whereas rendering the object normally may require multiple textures and multiple texture layers—changing texture flushes the hardware's texture cache and may require extra texture management from the driver, the API, or the application. Drawing the object each frame requires that it be lit each frame, even if the lighting has not changed, and as lighting techniques become more sophisticated, lighting becomes more expensive. Finally, there is usually plenty of application overhead associated with drawing an object, even before a single API call is made—using impostors can avoid much of that work.

The Whole Process

Impostoring involves a few key parts, which will be addressed separately:

- Rendering the impostor texture on the screen each frame. I call this "rendering" the impostor.
- Rendering the impostor's mesh at low speeds to the texture. I call this "updating" the impostor.
- Deciding when to update the impostor, and when to leave it.
- Driving the hardware efficiently.

Rendering the Impostor:

ONE:INVSRCALPHA

An impostor is essentially a color picture with an alpha channel that defines that picture's opacity in some way. Given that, there are basically two choices for this sort of blending: premultiplied alpha and "normal" blended alpha.

Normal alpha is the standard SRCALPHA:INVSRCALPHA style. The other style is premultiplied alpha— the ONE:INVSRCALPHA style.

Which one to use depends on which produces the correct result when rendering an alpha-blended object into an impostor, and then rendering the impostor to the screen. A pixel P is rendered to an impostor texture (which is cleared to black) to produce an impostor pixel I, and then that is rendered on top of the existing framebuffer pixel F, to produce result pixel R.

If using non-premultiplied alpha, the desired result is:

$$R = P \times P_\alpha + F \times (1 - P_\alpha)$$

The render to the impostor produces the result:

$$I = P \times P_\alpha + 0 \times (1 - P_\alpha)$$
$$I_\alpha = P_\alpha \times P_\alpha + 0 \times (1 - P_\alpha)$$

And rendering this to the framebuffer produces:

$$R = I \times I_\alpha + F \times (1 - I_\alpha)$$
$$= P \text{ x } P_a \text{ x } P_a \text{ x } P_a + F x \ (1 - P_a \text{ x } />_{„})$$

This is pretty useless, and very little like the desired result. With premultiplied alpha, the desired result is:

$$R = P + Fx(l\text{-}P_a)$$

The render to the impostor produces this result:

$$7 = P + O \text{ x } (1 \text{ - }_J P_a)$$
$$I_\alpha = P_\alpha + 0 \times (1 - P_\alpha)$$

Rendering to the framebuffer produces this:

$$R = I + F \times (1 - I_\alpha)$$
$$= P + F x (l - P_a)$$

which is perfect. Premultiplied alpha is not used by many apps, but it is fairly simple to adapt an existing engine to use it.

One thing to note is that it is now important to be precise about the alpha-channel result even when rendering opaque triangles, since the result will be written to the impostor, and will influence the result when die impostor is rendered to the screen. Non-alpha-blended (i.e., opaque) rendering to the impostor must ensure that the alpha result is 1, or the background will show through. Fortunately, this is fairly easy to do, but it does require more care with the alpha-channel result than is normally taken.

One problem with impostoring is that alpha-blend effects rendered into an impostor must be capable of expressing their effect within the premultiplied alpha scheme. This means that effects such as multiplicative color blends (basically, anything with a COLOR argument in the alpha-blend) will not work as intended, because the impostor has only a single alpha channel to express the amount of background to allow through. Fortunately, these sorts of effects are rarely used on objects that are suitable for impostoring. Note that this only applies to actual translucent effects — opaque multipass rendering that uses alpha blending to combine the passes (e.g., light maps, detail maps, etc.) will be fine, as long as the final alpha-channel value is carefully controlled.

Billboard Quad

The most obvious way to render the impostor is to simply render a quad representing the impostored object. This gives a perfect result as long as the neither the object or the camera move.

Unfortunately, pixels are not all the app has to worry about in a 3D scene; there is also depth to consider. A quad can only represent a single plane of depth, but the object it represents will cover multiple depths.

So, for the quad, the app needs to decide at what depth to draw. Unfortunately, for some of the most common applications, there is no single depth that is appropriate. As can be seen in Figure 5.7.1, our impostored hero is standing between two walls. Unfortunately, his feet are sticking through the near wall, and his head has vanished through the far wall. Quads are no good if an impostored object is going to be close to other objects. They may be quite good for flying objects that don't usually get too close to things, though, such as aircraft in a flight simulation.

Cuboid

The next approximation is a bounding box. Instead of a quad, the object-space bounding box of the object is drawn, with the impostor texture projected on it.

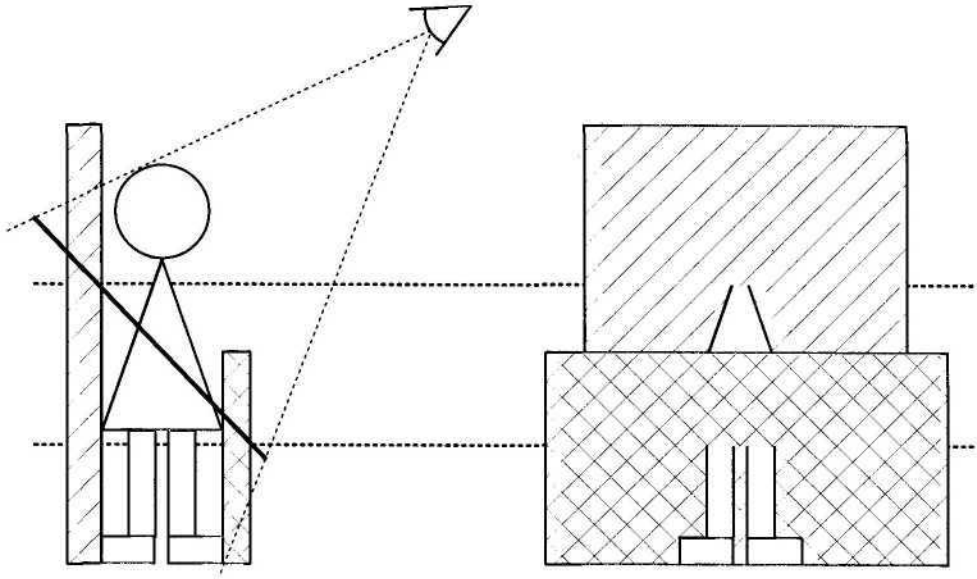

FIGURE 5.7.1 *Side view and rendered view of a billboard impostor showing Z-buffer problems.*

Because the bounding box is a real 3D object, its Z-buffer properties can be controlled far better than those of a screen-aligned quad. In particular, its Z-buffer properties are now independent of the camera position—they only depend on the object position and orientatipn, which is one less thing the app has to worry about.

Bounding boxes actually work quite well. Most things in everyday life fill their bounding boxes nicely, and artifacts from intersections between bounding boxes are rare. Note that only the front or back of the bounding box is drawn. Either can be used, but I recommend the front, because many objects fill their bounding boxes, so using the front side gives a parallax shift that is similar to the real object when the camera moves.

Although a bounding box is more complex than a quad, it's not much more complex—12 tris is not going to break the triangle budget now that typical scenes have hundreds of thousands of triangles. The increase in Z-buffering stability compared to a single quad is definitely worth the effort.

Bounding Object

There are plenty of objects for which a bounding box is not a good enough approximation, and using one leads to unnecessary Z-buffer clashes. Another factor to consider in choosing the shape of the impostor is parallax error. Most 3D scenes have a large number of elements that are stationary, but the camera will still be moving through that scene. A box painted with a picture of something on it is not going to look like that something for long when the camera starts moving. Although increasing

the rate at which the impostor is updated can help, this just burns pixel and triangle rates far faster for not all that much benefit—the eye very quickly notices the lack of parallax as it moves.

Using an impostor object that is closer to the real shape of the object, although still with a very low triangle count, can give a good improvement over a bounding box. The main restriction on the shape is that it needs to fully enclose the object; otherwise, the impostor image may be bigger onscreen than the impostor object being used, and the edge of the image will not get drawn. The other restriction is that the object must usually be convex to prevent self-sorting problems, because the impostor texture is drawn with alpha blending.

Image Warping

One of the problems with the impostor object is that it must be both convex and larger than the thing it represents. The former is required to prevent pixels being rendered twice, and the latter to prevent some pixels not being rendered at all. However, this inevitably means that the parallax as the object rotates (or the camera moves) is not going to be correct, because the impostor object is bound to be larger than the image it represents. Using a higher-tri impostor object can help reduce this in the case of entirely convex source objects. However, for nonconvex source objects, this does not noticeably improve matters—the impostor object must remain convex, and no amount of tris will allow it to match a concave object.

Another way to deal with this is to move the texture coordinates at each vertex each frame. The tri counts involved are fairly low, so a bit more work at each vertex is unlikely to hurt performance much. The principle is fairly simple—figure out where on the real object (and thus the impostor texture image) each impostor objects vertex lies when viewed from a certain angle. As this viewing angle changes, the texel that the vertex lies over will change. Therefore, for a new viewing angle, trace the line from the viewer through the impostor object vertex to the original object. Then work out which texel this part of the object was originally drawn to, and set the UV coordinates of this vertex accordingly.

Nice Theory, What Can We Get Away With?

This is expensive and fiddly to implement. It also has loads of problem cases, such as the vertex raytrace falling down a visually insignificant "chasm" in the object and, because of the low density of vertices in the impostor object, producing large warps over the image. Another problem is what to do with vertices at the edge of the impostor object that do not map to any part of the real object at all—what happens to them? Some type of interpolation seems to be needed from the visible edges of the object out to these vertices. This is the type of work that an app does not want to be doing at runtime, however few vertices it involves.

In practice, what I have found to be far simpler, and works just fine, is to give each vertex a "parallax factor"—the number of image texels to move per degree of viewer

movement. This is a factor usually tweaked by hand, and easily determines the vertex's texel coordinates at runtime. This factor is only done once for each impostor object vertex, and hand-tweaking around 8 to 10 vertices per object does not take long.

Alternatively, to generate these parallax values automatically for most real-world objects, find the nearest real-object vertex to each bounding object vertex. This distance is then proportional to the parallax factor required. The actual value depends on exactly how the factor is applied to the UV values, which depends on how the texture is mapped to the impostor object. For more complex objects, such as highly irregular, spiky, holed, or multilayered objects, it may still be better to hand-tweak the factors. This method finds the nearest vertex, but for these objects, it may be more appropriate to use some sort of average instead.

Even this simple method may not be possible because the objects are generated dynamically; for example, through animation. In this case, using a bounding cuboid and assuming an ellipsoid object inside works remarkably well for such a primitive approximation. It certainly works better than having no texture coordinate changes at all.

Update Heuristics

On each frame, for each impostor, the decision needs to be made whether to update it that frame. A number of factors contribute to this decision. In all these cases, some sort of screen-space error estimate is made for each factor, and the factors summed. If this sum is over a global factor (which may be static, object-specific in some way, or dynamic to try to maintain a certain frame rate), the impostor is updated.

Animation

Animation changes the appearance of the object, and at some point the error is going to grow too great. This can be quantified by a delta between the animation frame the impostor was rendered with, and the animation frame that would have been used if the object were not impostored. Doing this precisely requires the object to be animated each frame, even if an update is not needed. This can be quite a large part of the expense of rendering an object, and it is a good idea to try to avoid a complete animation step. The easiest way I have found is to do a preprocessing step on each animation and find the largest distance that any vertex moves during the animation. Divide by the length (in time) of the animation, and thus find a "maximum error per second" measure for the animation. This is easy to do in a brute-force way, and since it is a preprocessing step, this is perfectly reasonable.

Note that the human eye is absolutely superb at extracting humanoid movement from images just a few pixels high. Impostoring this motion, effectively reducing its effective frame rate, can be surprisingly noticeable, even at very slight levels. It is a good idea to have an extra bias on these animations that can place even more emphasis on them than simple mathematical screen-space error. This effectively rules out impostoring for anything but slight animations on distant objects.

Lighting

If the lighting changes significantly on the object, it will need to be updated. Since lighting systems are extremely varied between 3D engines, this requires fairly engine-specific routines to decide what the equivalent screen-space error is. Lighting a point at the center of the object using the six cardinal normals and comparing RGB differences between the current conditions and those when the impostor was created gives a fairly good idea of how the lighting has changed. Multiplying this by object size and dividing by the distance from the camera then gives a rough screen-space error.

Viewing Angle

Changing the viewing angle is probably the most obvious factor that decides an impostor update. Note that what is important is the vector from the camera to the object in object space. This will change when the object rotates, and also when the camera moves, both of which are important. The camera's direction of view is unimportant—unless an enormous field of view is used, the object does not change appearance much when the camera rotates, only when it moves.

Camera Distance

As well as the direction from the object to the camera, the distance between the two is also important. Although it does not change the actual appearance of the object, as the camera moves directly toward the object, the impostor texture gradually enlarges. After a while, it becomes obvious that this is just an image that is being enlarged by bilinear filtering, and not a real polygonal object, and so needs updating. This update will render to a larger texture, and so give more detail.

Game-Specific Heuristics

Many games also have specific heuristics that can be used to tweak these update rates. A common one for FPS games is that objects near the center of the view are usually the ones the player is looking at. These should get a slight boost to their update rates by dropping the acceptable screen error.

For mouse-driven games such as god games and RTS games, a similar tweak can be made to objects underneath the mouse cursor, for exactly the same reasons.

A distinction can also be made between "scenery" and "important" objects. Items that are purely in the scene to create an ambience, but are not usually involved in the gameplay, can be assigned a relatively large screen error. The player will not be examining them terribly closely—his or her attention is likely to be elsewhere. Therefore, these objects can be "more wrong" before the player notices the error.

Efficiency

The main efficiency hit on most cards is changing the render target. This causes flushing of many internal caches and states, and on some cards causes a flush of the entire

rendering pipeline. The main efficiency aim is to minimize these changes. The best way to do this is to wait until the end of the current scene, batching up the required updates to the impostor textures, rather than doing them as they are needed. Use large render targets, and at the end of the scene, pick the emptiest render target and render multiple impostor images to subregions of that texture.

Prediction

When updating an impostor, the current state of the object is rendered to the texture. This is then faded in over a few frames, and then preserved for a few frames more, before in turn being replaced by a newer render. This does mean that what is visible onscreen is always out of date.

This can be improved by doing some forward prediction of impostor state. The idea is to predict what the impostor is going to look like halfway through its lifetime. If an object is currently being updated every sixth frame, when updating the impostor, the state of the impostor (orientation, position, lighting, animation, etc.) should be forward-predicted by three frames.

With games such as first-person-shooters, objects in the world are basically split into two distinct categories: those that almost never move (walls, furniture, clutter), and those that move erratically (players). The movement of the players is notoriously hard to predict, and it is probably a waste of time trying to impostor players.

On the other hand, impostoring the scenery and furniture is a far more viable proposition. Prediction for them is trivial—they almost never move. And when they do move, they usually move under control of a player; in other words, erratically. The easiest thing is to simply disable impostoring for the duration of the movement.

For god games and Real-Time Strategy (RTS) games, the problems are similar, but the movement of the camera is very different. It is usually a bird's-eye view, and most of the time it is either static (while issuing orders to units), or moving at constant speed over the map to get to a different area. Small, erratic movements are rare, which is fortunate since these are extremely hard to predict. Prediction of moving objects can also be very useful in these games, since most of them are AI-controlled. Much of the time, the objects are either stationary or walking in straight lines to a destination, both of which are easy to predict. However, both camera movement and object movement can change abruptly, and when they do, the best thing is to flag the impostor for an update very soon, or even to temporarily disable impostoring altogether.

Summary

Impostoring is useful when trying to draw scenes with lots of fairly static objects in them. The raw triangle count will overwhelm any bus and graphics device that tries to render them at top detail, and progressive mesh methods can only do a certain amount to reduce the workload—texture changes and animation are extremely difficult to reduce in this way.

Impostoring is most effective on static objects some distance from the camera. Introducing this sort of clutter into games increases the visual quality substantially, especially since each object is still a real independent 3D object that players can interact with if they wish. It also allows key objects to be "hidden in plain sight" amongst a lot of other objects—something that has been extremely difficult to do with existing techniques and the limited number of objects available in a scene.

Even an implementation using a bounding box and some simple math produces good results for the incidental objects that are currently missing from games, but produce far more realistic scenes.