Hardware Accelerated Environment Deformation

Nicholas D. Marinelli

Abstract

Video games continue to become more and more interactive and dynamic as technology progresses. There are always more small items manipulated by physics, the AI becomes more and more adaptive to the player, and the areas to explore keep growing larger. Even though all these factors are growing, the player's interaction and effect with the world itself (rather than the things residing in the world) remains minimal. This paper presents a method to add real time deformation to the world around the player. Specifically, 3-dimensional bullet holes to replace the commonly accepted 2-dimensional decal.

Keywords: Interactive, Environment Deformation, Geometry Shaders

1. Introduction

Modern video games are plagued with having static environments. Despite advances in real-time physics which have allowed moveable objects in the world, the world itself is affected very little by the player. In games involving gunfire and explosions, this is particularly noticeable, as guns may leave bullet-hole decals, and explosions may leave black decals, but aside from these superficial marks, the world remains unchanged. This paper's goal is to use the latest features in graphics hardware and Direct3D 10 to implement a real-time, hardware accelerated environment deformation system.

The "Red Faction" series of games for the Playstation2 and PC implemented a primitive form of world deformation which was dubbed "GeoMod." Though this method did provide for a deformable world, it was not without its limits. All the holes created were roughly the same shape and size, regardless of the way they were hit. In addition, the number of holes was also greatly limited. After a short while, new holes would not appear. The method was also implemented completely in software, so if multiple holes were created concurrently, a great deal of slowdown could occur.

The aim of the paper is to eliminate some of the problems associated with the original "GeoMod." Geometry shaders will be used to implement this new method. The geometry shader is an intermediary step in the programmable pipeline, between the vertex shader and the pixel shader, and was recently introduced in Direct3D 10 [Blythe, 2006] (and extensions for OpenGL [NVIDIA, 2007]). As opposed to vertex shaders, which know only of single vertices being passed into them, and pixel shaders, which know only of single pixels being passed to them, a geometry shader is passed three vertices at a time, effectively giving it knowledge of an entire triangle. Knowing about an entire triangle easily allows the programmer to create new triangles that have a meaningful relationship with the original triangle. Figure 1 shows an example from the Direct3D 10 SDK in which the triangles of a model are separated from each other, demonstrating the level of control given to a programmer. In addition, to entire triangle manipulation, the geometry stream-out feature allows newly created geometry to be fed out from the geometry shader back to the beginning of the pipeline. This allows the GPU to cumulatively keep track of any new triangles and not force it to either recalculate every new triangle created with every frame, or force the CPU to keep track of the ever changing geometry.

Figure 1: The geometry shader allows for manipulation of entire triangles, as opposed to just vertices.

With this paper, a demo is presented, created with C++ and HLSL, which will allow a user to create the holes in the world around them. Since DirectX 10 geometry shaders, like any other shader, concentrate on local information, the very large holes created in "Red Faction" will be set aside, and the main goal will be to create 3D holes caused by bullets, rather than the standard 2D decals usually seen today. This paper removes the limitations on number of holes, and the greatly increases the speed at which they are calculated. In addition, the holes will vary in size and shape, repositioning and creating vertices based on a radial function centered at the point of impact. Research and consideration is also put into making the holes be dependant on adjacent holes, the material of the original surface, and the angle at which the shot intersects with the surface. Other methods are also considered which would allow very large hole spanning multiple triangles to be created, usually caused by some kind of explosion. These methods could be used to re-implement the global scene modification that "Red Faction" introduced, but incorporate all the advances that are made possible through the latest generation of graphics hardware.

2. Related Work

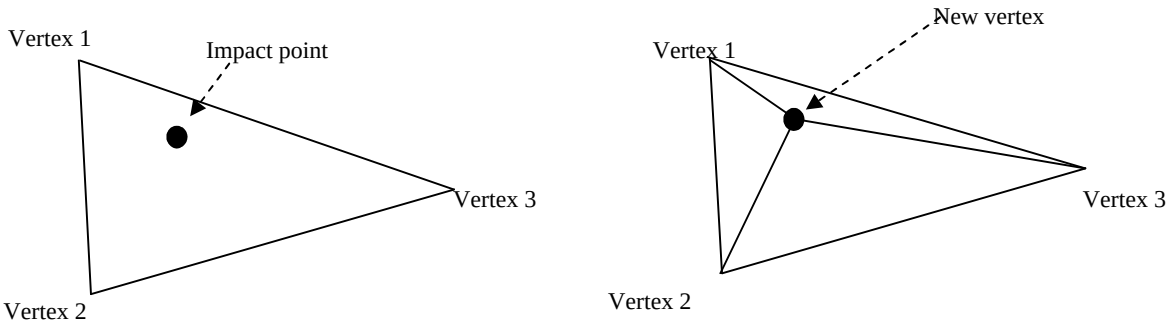

Figure 2: Three new triangles are created, using the impact point as a new vertex

The driving inspiration in this research was the work done by Volition Inc. in their "Red Faction" series of video games. To date, this is the only series of video games to feature the manner of terrain deformation that they created. As stated previously, though, the technology was limited. Unfortunately, most of their work is proprietary and was never submitted to any kind of graphics symposium such as SIGGRAPH or I3D.

Steven Workman [2006], however, saw the work done by Volition and aimed to create a realistic real-time cracking method for individual objects in a world. His approach incorporated many realistic concepts as he incorporates the physics of the forces interacting on the object (coupled with textures of object density), to calculate how an object is destructed. His methods of determining breakage points are very complex and thorough. It is possible to use the theory for his methods in the hardware, but not go through the great lengths of incorporating the most physically plausible breakage possible.

Workman took many ideas from O'Brien and Hodgins [1999] and optimized them to make them suitable for a real-time environment. The main difference between O'Brien's research and Workman's is that O'Brien calculates stress and likely fracture points while the simulation is running, while Workman pre-calculates them and stores them in a texture. Both O'Brien's and Workman's methods differed greatly from a method used to simulate construction [Kamat and Martinez 2003] in real time, which used a constantly updating 3D terrain database to keep track of the displacement of land during a construction process.

David Blythe originally showed off the power of Direct3D 10 at SIGGRAPH [2006]. He demonstrated how the power of geometry shaders could be used for various environmental effects. The stream-out feature, which could feed back to the vertex shader new geometry produced by the geometry shader, was also a key idea in this new technology. Research yielded other examples [Tariq 2006, Blythe 2006] on specific syntax and usage of Direct3D 10 technology.

On a simpler approach, which implements a makeshift approach to the problem of replacing 2d bullet decals, Woodhouse [2003] uses stencil buffers to not draw areas where decals are, and then project the decals texture behind that area. Though effective for the older hardware that it would work well with (voodoo 3 and up), the latest graphics hardware was designed specifically to handle this task in a better fashion, and using the geometry shader to do that is exactly what this paper is intended to show.

Displacement maps were a concept first shown in "Shade Trees" [Cook, 1984]. By storing a displacement amount in a texture on the surface, the vertices on the surface could be manipulated. The RenderMan Companion [Upstill, 1989] demonstrated the use of displacement maps to allow patches of displaced material. Though each patch can create a believable effect on its own, the patches cannot interact with each other. When two patches overlap, they will not magnify the displacement in either patch, and may even simply overlap the existing patch.

3. Implementation

This paper implements a demonstration of the methods described using a sample Direct3D application which loads a model and allows the user to click on any point of that model. The normalized screen coordinates of that mouse click are then passed to the shader. Based on these coordinates, the shader is able to create a ray and determine if the user clicked on a specific polygon. This process is more commonly known as "picking." Any geometry can be loaded into the program, but for simplicity's sake, a wall model is used to fully demonstrate how the method could be best used.

3.1 Intersected Triangle Manipulation

The primary function of the geometry shader program implemented takes in the normalized screen coordinates as parameters passed in by the program. It then uses these coordinates to trace a ray and determine if the mouse click intersects the current triangle being run through the shader. If the ray intersects the triangle, the intersection point is known as the "impact point," and a new vertex is created at that point. This vertex is then displaced slightly into the model, to simulate how far a bullet would penetrate. The vertex is displaced in the opposite direction of the normal of the surface. Three new triangles are then created, each using two of the original vertices and the newly created vertex, and the original triangle is discarded, as it would block the hole if it were kept. Figure 1 illustrates the three new triangles being created due to the introduction of the impact vertex. In addition to the impact point being displaced towards the inside of the model, the original three vertices are displaced at as well.

Based on how close a vertex is to the impact point, the vertex will be recessed farther into the model. The closer to the ray is to the intersection point, the greater the effect becomes. This is determined using a Gaussian fall-off. With the distance between the impact point and a given vertex in the current triangle, the Gaussian function is used to make each vertex be displaced in the same direction as the impact point, but with a displacement distance equal to a fraction of the impact point's displacement distance.

3.2 Surrounding Triangle Manipulation

In addition to the triangle that is actually hit by the projectile, the effect should be felt by surrounding triangles as well. Vertices were originally thought to be shared between adjacent triangles by default. This turned out to not be the case. The intended effect could not be created by passing regular triangle lists to the geometry shader. Adjacency information is required to be included in the triangle list for this to be possible. Unfortunately, there was not enough time to implement this feature. The original theory was that since the impacted triangle's vertices are displaced, the immediately surrounding triangles should automatically affected by the initial vertex displacement done on the triangle containing the impact point, as these triangles share the displaced vertices with that original triangle. The geometry shader, however, did not act as expected, and the entire bullet hole became displaced, rather than pulling the surrounding triangles with it. If adjacency information was included with the incoming triangle list, then the shader would have access to the adjacent triangles, and would be able to deform along with the impacted triangle.

An optional feature is possible, but not implemented, which would allow non-adjacent triangles to be affected by the impact. This would involve taking the proximity of the triangle to the ray of the projectile into account. This is discussed in the future work section.

3.3 Geometry stream-out

Recalculating the sum of all changes made to the geometry with ever hole made would not only be inefficient in processing time on the GPU, it would also bog down the CPU a great deal as it would need to keep track of what holes have been created and in what order. This is the primary motivation behind using the stream-out feature associated with geometry shaders. Using the stream-out, the newly created geometry can be saved, and the next set of calculations can be done on the newly saved geometry, rather than having to recalculate every hole on every frame. Figure 2 shows the Direct3D 10 pipeline. Triangles may be streamed out after the geometry shader stage into vertex buffers. These buffers are then recycled into the input assembler stage.

The stream out stage takes place immediately after the geometry shader (or vertex shader if one so desires). It can either be fed directly back into the input assembler, or the user can declare in their code one vertex buffer meant for streaming out (of the GPU) and one buffer meant for drawing from. These buffers cannot be the same and must be designated as being used with stream-out. Work was initially started to bring the stream-out data back onto the CPU. Though the buffer is returned to the CPU, no extra manipulation or calculation is necessary, as it has all been done on the CPU. The CPU need only pass the vertex buffer back into the GPU. As the geometry shader runs, and more holes are created, this newly formed geometry is streamed out. When it is streamed back into the GPU, the geometry shader is now performing its intersection tests against the previously deformed geometry, rather than the original. As this process loops, it allows more holes to be created. Due to time restrictions, and very poor Direct3D documentation, however, the stream-out feature was unable to be completed.

4. Results

The results of the primary research goal were partially successful. Given any triangle, it could divide it into three and recess it into the surface. Attempts to implement surrounding triangle manipulation and persistent deformation through stream-out were unsuccessful, as previously stated. Figure 4 demonstrates how the entire hole is displaced due to lack of adjacency information, while figure 5 performs the impact on every surface, but without initial vertex displacement.

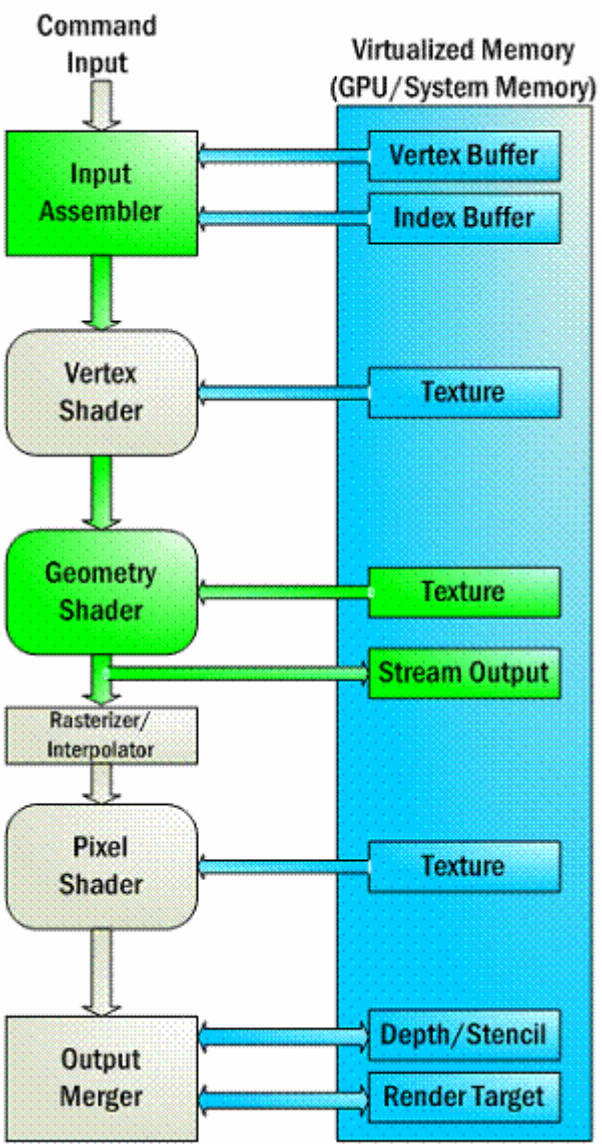

Figure 3: The Direct3D pipeline. The geometry shader stage and stream output are the two biggest additions.

5. Conclusions

Due to time constraints and a lack of understand of the usage of Direct3D 10, this research fell short of expectations. Though considerable thought was put into every stage, the actual use of the D3D API proved to be far more difficult than expected.

In addition, the method has a number of shortcomings. Since it divides the impacted triangle into 3 parts, it requires a highly tessellated surface to reach the desired effect. In addition, the hole itself will always be triangular. Though the vertices of the triangle move, affecting the triangles around it, the hole itself remains a triangular hole. Though this research does address the

issue of 3-dimensional bullet holes, there is great room for improvement.
Direct3D 10 is a very new technology as well. Documentation and examples for even the most common functions is currently near impossible to find, which adds an immense amount of difficulty in attempting to implement this method. OpenGL should have been given more consideration, as geometry shaders are possible with it. Direct3D 10's stream out feature is very poorly optimized, as well. This, in conjunction with the still very young hardware in the latest graphics hardware, will cause a great deal of processing overhead until the technology becomes more widespread.

6. Future work

There are a number of possible features that could be added onto this research in order to address the shortcomings discussed previously. To rely less on an initially highly tessellated surface, upon impact, the surface could be recursively subdivided into smaller sets of triangles, until the desired tessellation is achieved. This would allow for the effect to be retained, removing the burden of making the surface work will with the shader off of the artist designing the geometry.
In the same vein as "Red Faction," explosions could generate larger holes than a bullet would. This could potentially be done by adding more attributes to the projectile, and if the impacting projectile creates an explosion, vertices not in the impacted triangle would also be put into consideration. This could potentially be accomplished by determining how close the projectile passes to the triangle through a barycentric coordinate test. Using the same Gaussian fall-off, the vertices would be displaced in a direction away from the path of the projectile.
A non-triangular hole would add more realism to the shader. By creating several new vertices inside the impacted triangle, instead of just one, a hole of a more realistic shape would be created in the triangle. Realistic cracking algorithms could also be implemented on top of this method. Since Workman's algorithm [2006] was based on textures of weakness in a material, these textures could be considered during the geometry shader stage, and the shape of the hole created accordingly.
It would also be a true demonstration of this method's purpose, if the shader could be generalized to be used outside of a simple Direct3D window, and used as an actual material of a surface in a video game.

References

1. David Blythe, "The Direct3D 10 system," SIGGRAPH 2006, pp. 724 – 734.
2. Vineet R. Kamat, and Julio C. Martinez, "Automated Generation of Large-Scale Dynamic Terrain in 3D Animation of Simulated Construction Processes," CONVR2003, pp. 63-76.
3. Sarah Tariq, "DirectX10 Effects," SIGGRAPH 2006, http://developer.download.nvidia.com/presentations/2006/siggraph/dx10-effects-siggraph-06.pdf.
4. David Blythe, "Direct3D 10," http://www.csee.umbc.edu/~olano/s2006c03/ch02.pdf, 2006.
5. NVIDIA, "NVIDIA OpenGL Extension Specifications for the GeForce 8 Series Architecture (G8x)," http://developer.download.nvidia.com/opengl/specs/g80specs.pdf, 2007.
6. Steven Workman, "A Cracking Algorithm for Destructible 3D Objects," http://www.dcs.shef.ac.uk/~aca03sw/report.pdf, 2006.
7. James F. O'Brien, and Jessica K. Hodgins, "Graphical modeling and animation of brittle fracture," SIGGRAPH '99, pp. 137-146.
8. Francis Woodhouse, "3D Decals," http://www.gamedev.net/reference/articles/article1986.asp, 2003.
9. Robert L. Cook, "Shade Trees," SIGGRAPH 1984, pp. 223-231.
10. Steve Upstill, "RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics", 1989

Figure 4: Displacing the original triangles vertices moves the entire impact crater

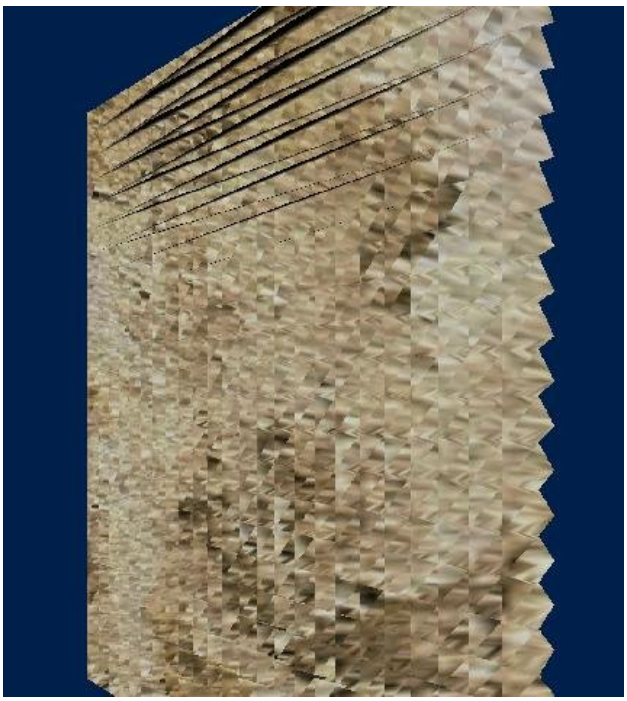

Figure 5: Demonstration of each triangle being impacted. (Top portion a artifact of improper stream-out)