Synthesis of Methods Relating to Real-Time Deformable Terrains

Andrew Flower[*]
Department of Computer Science
University of Cape Town

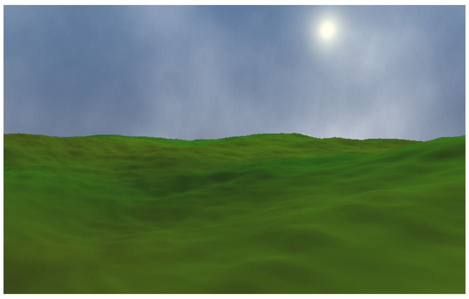

Figure 1: *A heightmap terrain generated using Perlin Noise.*

Abstract

This paper reviews current techniques for the internal storage and efficient rendering of 3D surfaces, specifically terrains. The aim is to identify viable algorithms allowing the incorporation of real-time deformable terrains into modern 3D games. Detailing-techniques covered include displacement and bump mapping. In order to provide dense meshes for displacement, a number of tessellation schemes are covered including Catmull-Clark subdivision surfaces. We review methods that could allow deformations due to any arbitrary impressions in the terrain such as footprints. Only GPU solutions are considered in the paper as the algorithms are highly parallel. To handle the storage and rendering of the possibly large terrain data, some LOD techniques are presented. The techniques most suited to terrain deformation are a combination of adaptive tessellation and displacement mapping where regions of detail are subdivided on the GPU before displacement occurs. Geometry clipmaps provide an efficient LOD scheme as do selective refinement implementations of progressive meshes.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

Keywords: subdivision surfaces, programmable shaders, geometry shaders, displacement maps, deformable terrain

1 Introduction

Geometry meshes have been an essential, fundamental part of 3D computer games since their inception. In fact, almost every object in a virtual 3D world is represented by some mesh. For many years the geometry of 3D environments remained static, unaffected by player interactions. Interactive objects have since been incorporated into modern games in increasing numbers, predominantly providing collision detection and response and on the rarer occasion, destructibility. Between static and destructible meshes there is the option of deformable meshes. This paper reviews graphics programming techniques that could benefit an implementation of real-time, deformable terrains for use in computer games. The focus is high-detail deformations such as footprints in snow or indentations of arbitrary shapes.

[*]e-mail: aflower@cs.uct.ac.za

Deformable terrain has not been implemented widely in 3D games, notable exceptions being LucasArts' *Fracture* or EA's *Battlefield: Bad Company*. Reasons for this include the computational cost involved and the ability to maintain consistency across the network in multiplayer games. In the past terrains were stored as explicit static mesh models but nowadays with the capability of programmable Vertex Shaders, terrains are being generated on-the-fly using heightmaps. Heightmap terrains are generated using a technique introduced by *Robert L. Cook* [Cook 1984] as *Displacement Mapping*. The fundamental idea is that a planar, regular mesh comprising vertices and their normals can be offset at each vertex by magnitudes specified in a separate structure known as the *Displacement Map*. When Cook wrote the paper there were no programmable shaders, nor would anyone have considered real-time rendering at this level. Nevertheless, his methods are still widely used today. This topic will be covered in a later section.

Cook's technique of displacement mapping was actually an extension [Cook 1984] of the image-based technique *Bump Mapping*, proposed by *James F. Blinn* [Blinn 1978]. There are a number of bump-mapping techniques that are discussed in sections below. The central idea to avoid modifying geometry and to rather create a visual illusion of irregularities on a surface by controlling the lighting reflection. This is usually done by storing normals and other information in textures called *bump maps*. Bump mapping is a viable solution for adding fine detail to surfaces without massive computational costs.

Since the advent of programmable shaders many algorithms are now possible in real-time. Shaders do however, have limitations and thus many algorithms need to be redesigned to efficiently use the graphics hardware. The latest Graphics Hardware gives rise to a *Tessellation* stage in the processing pipeline. We will only be considering Shader Model 4.0 corresponding to DirectX 10.x and OpenGL3.x so that techniques will not be limited to the most expensive, new cards.

The next section will explain more about the GPU, its programmability and memory storage options. Section 3 will describe methods of mesh representation in greater depth (with the focus on adding detail), including tessellation techniques. In Section 4, image-based detailing is considered. Level-of-Detail techniques are covered in Section 5 and the conclusion summarises the most important techniques covered.

2 The GPU

The *Graphics Processing Unit* has become a very powerful parallel processor with the new NVIDIA chips containing 480 cores (scalar processors) [NVIDIA 2010]. It is this large number of highly capable parallel cores that allows the graphics pipeline to process data so quickly. The same operation is executed on multiple vertices (transform etc.) or fragments (shading etc.). This architecture is known as Single-Instruction-Multiple-Data (SIMD). In the past decade the fixed-function pipeline of graphics cards has become programmable, allowing the developer to specify generic instructions to apply to pipeline elements.

2.1 GPGPU

Once people realised the computation power of GPUs, they started using shaders in new and interesting ways. The computation-to-price ratio is outstanding and is a great incentive for scientists to invest in powerful GPUs in order to run simulations [Green 2005a]. Many algorithms were implemented on the GPU such as N-body simulations, large matrix multiplications and finite difference schemes. Although these produced much faster results, their implementations were non-intuitive and very finicky. Since then NVIDIA has released their GPGPU framework called CUDA. CUDA creates an abstraction of the GPU hardware so that developers may treat the hardware as a collection of SIMD processors which execute generic kernels rather than hacking graphic shaders together. Many of the graphical effects use a GPGPU-esque approach and CUDA can be used in conjunction with OpenGL to simulate and render fluid simulations or particle simulations.

The geometry shader can be used as a stream processor [Diard 2007] performing calculations on vertices containing arbitrary data and writing variable-length results to feedback buffers or arbitrary texture locations. This flexibility enables a wide range of algorithms to be implemented on the GPU and is useful to multi-pass techniques for writing intermediate data.

2.2 Memory

Modern GPUs offer a significant amount of memory. The memory can be used directly in the CUDA API but only accessible implicitly in OpenGL via constructs like textures and VBOs (Vertex Buffer Objects). Storing geometry and images on the GPU is a favoured approach as it minimises CPU to GPU bus transfer which has high latency. This is enforced in OpenGL 3.0 with the deprecation of immediate mode and the fixed function pipeline. Although its purpose is to store graphical data, VBOs can be treated as regular memory arrays.

Textures are available to all three shader stages for sampling however texture formats, addressing modes and interpolation may be limited in the earlier shader stages. It is also possible to dynamically write/render to texture objects from the fragment shader with the use of FrameBuffer Objects (FBOs) [Green 2005b]. Using transform feedback, the geometry shader has the ability to output variable length data to VBOs, skipping the fragment shader and rendering stage.

Textures can be used to store mesh data as well, where the x,y,z coordinates are stored as r,g,b. Such textures are known as *Geometry Images* as seen in [Hernández and Rudomin 2006]. An advantage of GIs is that level-of-detail (LOD) meshes can easily be created using mipmaps of the GI.

The most difficult part of implementing algorithms such as tessellation in graphics shaders is the mapping of data structures to textures and buffers. Using the above techniques it possible to treat GPU memory as regular memory.

3 Mesh Representation

Now that the processing and storage capabilities of GPU hardware are understood, a few popular methods of surface deformation and refinement will be presented. These algorithms are performed on the GPU to decrease the usage of CPU-GPU bus bandwidth and permit dynamic surfaces.

3.1 Displacement Mapping

Displacement maps [Cook 1984] are textures storing a height field. Such height fields can be used to display fine detail on geometry or to represent larger coarser meshes. The displacement map will usually use the same texture coordinates as the mesh's other texture maps. Each texel contains a value indicating the magnitude of the corresponding vertex's displacement. These maps can be generated in a number of ways. One method is to use Perlin noise allowing control of different frequency layers. Another popular method is the Diamond-Square algorithm [Fournier et al. 1982] which is a fractal-based approach. These maps appears as cloudy grayscale images if viewed as images. A given vertex will lookup its corresponding displacement in the map using its texture coordinates, and the vertex would then be translated accordingly along its normal. A problem with displacement maps is that the mesh needs to be dense in order for the displaced surface to appear smooth, otherwise the piecewise linear triangles/quads become noticeable.

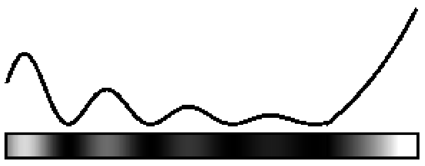

FIGURE 3.1: *A 1D example of a displacement map and the surface it represents.*

Once the surface is displaced, the vertex normals are no longer correct and must be recalculated if they are needed for lighting. We know from calculus that the normal at any point of a smooth surface can be computed as the cross-product of the tangent and binormal vectors, as this is their definition. Thus the displacement surface's normal can be calculated by choosing the tangent and binormal vectors to be the changes in the u and v texture coordinates respectively. [Szirmay-Kalos and Umenhoffer 2008] give an in-depth look at displacement mapping and these calculations.

If the original mesh is just a horizontal plane, then the calculated normal can be used as the normal, otherwise it needs to be combined with the normal of the base mesh to find the actual normal. To avoid the complicated combination, lighting calculations are usually performed in Tangent Space [Szirmay-Kalos and Umenhoffer 2008]. This space is defined as the space with tangent, binormal and normal as the basis vectors. The paper mentioned covers the math involved. Alternatively, instead of calculating the normals, they can be stored in a normal map texture to save computation.

Displacement mapping is a very useful technique for adding surface detail to dense meshes, coarse terrain meshes or for dynamic surfaces such as ocean waves. Real-time deformation is thus also easily possible by simply editing the displacement map. For uniformly dense meshes rendering becomes an expensive process in which case image techniques like bump mapping are used.

3.2 Tessellation

Tessellation is a process whereby a given surface is split up into tiles that fully cover the original surface without overlaps. In computer graphics the tiles are usually quads or triangles. Minimising vertex count in meshes is critical to increasing flow through the graphics pipeline. The goal for tessellation in modern 3D applications is to produce smooth 3D objects on-the-fly from coarse meshes. This was a very difficult and expensive multi-pass procedure before the introduction of Shader Model 4.0 and the Geometry Shader. Although the geometry shader is limited in comparison to the modern tessellation pipeline it still makes tessellation possible.

Tessellation is an inviting concept for displacement mapping. If tessellation can be done in an adaptive manner, such that areas of great displacement are more dense, displacement mapping will become a more prominent technique for displaying detail [Moule and McCool 2002].

3.2.1 Subdivision Surfaces

Like most graphics concepts, the generation of smooth surfaces has been a research topic for many years. A number of mathematical surface representations have been considered such as tensor product surfaces and b-spline surfaces. Evaluating these parametric surfaces is an expensive process [Huang et al. 2007] and thus doing it in real-time is out of the question. Catmull and Clark [Catmull and Clark 1998] devised a method for generating approximations to these surfaces with adjustable accuracy, a method termed *Subdivision*. A subdivision surface is defined by a coarse (usually rectangular) mesh known as the *control mesh*. It is akin to the control points in parametric surfaces or splines. Each of the faces can be subdivided according to rules specified by a *refinement scheme* [Patney et al. 2009]. This refinement process is performed recursively until the desired level of detail is reached.

Catmull-Clark subdivision converges quickly to a bicubic b-spline limit surface after few iterations. The benefits of this method are quick convergence, simplicity of implementation and the fact that each patch can be generated independently with just local information. Less memory is required to store the control mesh than would be needed for usual smoother meshes. The locality of the tessellation allows for relatively easy parallelization. In order to subdivide a quadrilateral, the quad's vertices and the 1-ring neighbourhood of vertices are required.

Currently two major schemes exist for computing subdivision on GPUs [Kazakov 2007]. One involves multiple passes with intermediate results being stored in graphics memory whilst the other performs direct evaluation in a single pass but requires texture lookup tables for tessellation patterns. The algorithm can be simplified if constrained to input control meshes with vertex valences of 4 (quads).

Another approach is to parallelise each step in a breadth-first approach [Patney et al. 2009]. The different stages are facepoint generation, edgepoint generation, vertex updating and rendering. This approach is more suited to a CUDA implementation where each stage can be shared by all cores. The vertex dependencies of faces on arbitrary meshes is non-deterministic however, thus making memory-coalescing difficult.

A final evaluation method of Catmull-Clark surfaces involves storing the basis functions in a texture along with corresponding 1-ring control points [Huang et al. 2007]. This would allow exact evaluation of the surface after lookups.

3.2.2 Alternatives

Phong Tessellation [Boubekeur and Alexa 2008] is a simple alternative to subdivision surfaces. The purpose of the algorithm is to create smooth silhouettes and contours rather than generating formal parametric surface approximations. The algorithm simply requires all the triangle vertices with their normals. From these attributes a new internal point is generated by projecting the point to the vertices' tangent planes and then interpolating using barycentric coordinates. The process is similar to gouraud shading or Phong's normal interpolation.

A second alternative is to use GIs (Geometry Images) with mipmapped LOD [Hernández and Rudomin 2006]. The mesh data is stored in textures with mipmaps storing less dense meshes. The mipmaps can be sampled by shaders so that the desired detail level is met.

As a final technique to consider for mesh representation is the *Progressive mesh* scheme [Hoppe 1996]. This scheme allows a complex, detailed mesh to be simplified to a coarser mesh of face-count orders of magnitude lower. During simplification edges in the mesh are collapsed by a controlled process and recorded in a list. The *edge collapse* transformations are invertible and thus the original mesh can be obtained by performing the inverse *vertex splits*. Select refinement can be performed by applying only the vertex splits occurring in desired areas of a mesh. In addition to selective refinement, the LOD possibilities of progressive meshes are effective and will be covered in section 5.

4 Bump Mapping

Bumping mapping, as introduced by Jim Blinn [Blinn 1978], involves varying normals across a surface in such a way that it appears to have bumps on it when lit. Bump mapping is sometimes used synonymously with *normal mapping*, and on the odd occasion refers to an image heightmap. In Autodesk Maya, for instance, the user can supply a grayscale heightmap as the bumpmap. The software implicitly converts the to a normal map and then applies normal mapping. Bump mapping is used as an alternative to displacement mapping because it much less expensive for producing fine detail.

4.1 Normal Mapping

This was the technique introduced by Jim Blinn where by a vector field/map of normals is supplied with a mesh. The values in this *normal map* are merely the normals along the surface that should be modeled. The texture coordinates of this mesh are used to index the normal map and locate the corresponding normal. Using this "fake" normal, the usual lighting calculations are performed. In Blinn's paper he demonstrates the bumps of an orange skin applied to a sphere. Traditional texture maps and displacement maps can be used in conjunction with Normal maps to create composite effects. Normal mapping provides a computationally inexpensive means to displaying fine detail on surfaces.

4.2 Parallax Mapping

This technique [Kaneko et al. 2001] extends the notions of texture and normal mapping by considering the parallax effect. Consider your line of sight to a swimming pool floor. Imagine now that the floor is actually the triangle to render and the water surface is the bumpmap surface. The point on the water surface that the view vector intercects is not orthogonally above the floor point that it intersects. This is the parallax effect. Traditional normal mapping would use lookup the height using the floor point's texture coordinates which is not the height of the surface at intersection. Instead parallax mapping will locate the point of intersection and its corresponding texture coordinates. These coordinates are used instead for looking up heights, colours and normals from the relevant texture maps.

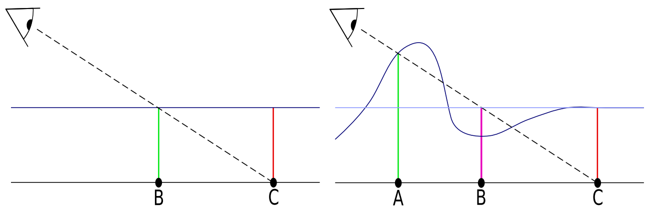

FIGURE 4.1: *Illustrates the principle behind parallax mapping. C is the projection used in normal mapping, B is the naïve parallax mapping using a constant surface, whilst A demonstrate the correct solution.*

The original naïve parallax mapping approach uses a constant surface height approximation to simplify calculations. Although this is an approximation, it still yields better results than normal mapping. Other approaches exist for locating the actual surface intersection point with varying accuracy and computational cost [Szirmay-Kalos and Umenhoffer 2008]. These techniques include iterative search, binary search, secant methods and others as heuristic approaches that will find the closest intersection sometimes. To ensure that the closest intersection is found expensive searches are required.

4.3 Relief Mapping

Relief mapping [Oliveira et al. 2000] is another view-dependent texture mapping technique that can be used as an alternative to parallax mapping. It produces more accurate results and supports normal mapping, self-occlusion, self-shadowing and silhouettes [Policarpo et al. 2005]. Unfortunately this technique requires a considerable amount of per-pixel computation and is not common in 3D games yet.

5 Level of Detail

Rendering terrains can result in massive number of vertices being rendered with each face undergoing displacements and texture techniques. To allow for scalability to large terrains, the algorithms need to incorporate some sort of LOD (Level of Detail) system. Initial methods for doing this would include backface and frustum culling but as maps become become more detailed, these methods are not enough. The LOD technique is greatly dependent on the mesh representation.

If the mesh is represented using geometry images, LOD can be implemented by creating mipmaps of the GI and its normal GI. Along with these textures, a selector texture [Hernández and Rudomin 2006] is constructed preprocess to be used to map distance-to-vertex to mipmap-level which allows real-time LOD choice. If the vertex shader supports non-power-of-two texture sampling (for the selector map), it may be possible to perform rendering in a single pass.

A possible LOD scheme for displacement maps involves storing displacement intervals rather than just magnitude [Moule and McCool 2002]. This interval is the minimum and maximum of the displacement function that is located in the particular lookup cell. For lower levels, the intervals are merged with neighbouring cells so that the new interval is the union of the two neighbour intervals. The next step is the selection of LOD at runtime. It could simply select the level that fully contains the maximum and minimum of the displacement function that region. This generates an unnecessary amount of detail. A better approach should be used for selection of levels of detail in each primitive as shown by McCool and Moule [Moule and McCool 2002].

A common problem with meshes having different levels of detail is the obvious 'pop' observed when they change levels. Progressive meshes [Hoppe 1996] offer the ideal concept of continuous LOD. Because the original mesh can be obtained by applying the vertex splits to the simplified mesh in sequence, it allows LOD of a mesh to increase one face at a time. This is a good solution to the infamous popping artefact, instead producing smooth visual transitions known as *geomorphs*. This LOD property can be combined with the scheme's ability to do selective refinement and may thus prove useful in large deformable terrains.

A final technique is that of Losasso and Hoppe, *Geometry Clipmaps* [Losasso and Hoppe 2004]. This involves maintaining an LOD pyramid around the camera. Regular grids of different density are nested around the camera. Each grid has the same number of tiles but the tile size is halved in each dimension for subsequent layers. This means that lower LOD grids extend further in world space than the higher density grids. If the grid is maintained in eye-space, grid recomputation is not needed and sampling of the textures can be done according to a moving offset. This technique permits very large data sets and even supports compressed textures.

6 Conclusion

The presented techniques have demonstrated many different approaches to terrain representation. The goal of designing a system for real-time deformable terrain is therefore a possibility. Due to the ease of changing displacement maps, they are an obvious choice for such a system. However their dependence on high mesh density means that such a system would require a method of adaptive tessellation to provide a scalable solution. Adaptive tessellation is certainly possible as seen in methods used by McCool and Moule [Moule and McCool 2002] and Patney[Patney et al. 2009].

The programmability of the graphics pipeline and the ability to use GPU memory in a flexible way permits implementations of these algorithms that may not have been possible with previous limitations. The addition of a simple LOD scheme will improve rendering time and lower the hardware requirements for support of the cheaper DirectX 10 compatible cards. Which tessellation scheme to use will require experimentation and comparison starting with the simple Phong Tessellation [Boubekeur and Alexa 2008] and progressing to Progressive Mesh or Catmull-Clark implementations. Comparisons of speed, memory usage and compatibility should be considered.

References

BLINN, J. F. 1978. Simulation of wrinkled surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 286–292.

BOUBEKEUR, T., AND ALEXA, M. 2008. Phong tessellation. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, ACM, New York, NY, USA, 1–5.

CATMULL, E., AND CLARK, J. 1998. Recursively generated b-spline surfaces on arbitrary topological meshes. 183–188.

COOK, R. L. 1984. Shade trees. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 223–231.

DIARD, F. 2007. *Using the Geometry Shader for Compact and Variable-Length GPU Feedback*. Addison-Wesley.

FOURNIER, A., FUSSELL, D., AND CARPENTER, L. 1982. Computer rendering of stochastic models. *Commun. ACM 25*, 6, 371–384.

GREEN, S. 2005. *Genereal-Purpose Computation on GPUs: A Primer*. Addison-Wesley.

GREEN, S. 2005. The opengl framebuffer object extension. In *GameDevelopers Conference*, NVIDIA, San Fransisco.

HERNÁNDEZ, B., AND RUDOMIN, I. 2006. Simple dynamic lod for geometry images. In *GRAPHITE '06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, ACM, New York, NY, USA, 157–163.

HOPPE, H. 1996. Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 99–108.

HUANG, X., LI, S., AND WANG, G. 2007. A gpu based interactive modeling approach to designing fine level features. In *GI '07: Proceedings of Graphics Interface 2007*, ACM, New York, NY, USA, 305–311.

KANEKO, T., TAKAHEI, T., INAMI, M., KAWAKAMI, N., YANAGIDA, Y., MAEDA, T., AND TACHI, S. 2001. Detailed shape representation with parallax mapping. In *In Proceedings of the ICAT 2001*, 205–208.

KAZAKOV, M. 2007. Catmull-clark subdivision for geometry shaders. In *AFRIGRAPH '07: Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, ACM, New York, NY, USA, 77–84.

LOSASSO, F., AND HOPPE, H. 2004. Geometry clipmaps: terrain rendering using nested regular grids. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, ACM, New York, NY, USA, 769–776.

MOULE, K., AND MCCOOL, M. D. 2002. Efficient bounded adaptive tessellation of displacement maps. In *In Graphics Interface*, 171–180.

NVIDIA. 2010. Geforce gtx480 specifications. NVIDIA Corporation.

OLIVEIRA, M. M., BISHOP, G., AND MCALLISTER, D. 2000. Relief texture mapping. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 359–368.

PATNEY, A., EBEIDA, M. S., AND OWENS, J. D. 2009. Parallel view-dependent tessellation of catmull-clark subdivision surfaces. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, ACM, New York, NY, USA, 99–108.

POLICARPO, F., OLIVEIRA, M. M., AND COMBA, JO A. L. D. 2005. Real-time relief mapping on arbitrary polygonal surfaces. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, ACM, New York, NY, USA, 935–935.

ROST, R. J. 2009. *OpenGL Shading Language*. Addison-Wesley.

SZIRMAY-KALOS, L., AND UMENHOFFER, T. 2008. Displacement mapping on the gpu - state of the art. *Comput. Graph. Forum 27*, 6, 1567–1592.