Virtual Texture: A Large Area Raster Resource for the GPU

Anton Ephanov, Chris Coleman
MultiGen-Paradigm
Richardson, TX

ABSTRACT

Texture mapping has traditionally played a key role in real-time 3D computer graphics, where it is used as the primary means for adding realism to the scene. Programmable Graphics Processor Units (GPUs) allow techniques which utilize textures as a generic look-up resource, thereby allowing textures to represent non-visual information about the database area, such as spectral data at multiple wave bands, thermal data, normal displacement maps for improved terrain shading, digital elevation maps for the terrain shape, or material-encoded maps for parametric approaches to providing dense organic scene content.

For large area visual simulations, the total amount of raster information for a database typically far exceeds available computer and video memory. Therefore, the image generator subsystem faces a substantial data management problem where it has to provide and combine heterogeneous resources (textures containing various data formats) to achieve the desired image quality and real-time performance characteristics. The data management (streaming) aspect of large-area coverage is equally important. The approach should be inherently efficient to address the challenges of modern combat training, where sometimes only a limited bandwidth is available for on-demand database streaming.

The paper presents a novel approach, called Virtual Texture, that addresses the challenges of utilizing huge amounts of raster data on the programmable graphics pipeline. The Virtual Texture formulation satisfies the key requirements of managing the data at deterministic real-time frame rates, while at the same time behaving as if it were a regular texture available on any texturing unit. The latter aspect of the formulation makes it possible to utilize multiple Virtual Textures in the context of any GPU-based technique or a specific vendor extension (such as SGI's detail texture), thereby significantly expanding its application domain. The paper includes examples of techniques where Virtual Texture has been used successfully to address simulation demands.

ABOUT THE AUTHORS

Anton Ephanov received B.S. and M.S. degrees in Mathematics and Mechanics from Moscow State University (Moscow, Russia) and a Ph.D. degree in Mechanical Engineering specializing in Robotics from Southern Methodist University (Dallas, TX). He is currently the Principal Architect for the Vega Prime run-time product at MultiGen-Paradigm.

Chris Coleman graduated summa cum laude with a B.S. in Computer Science from Texas A&M University (College Station, TX) and is currently working towards a M.S. in Visualization Sciences also from Texas A&M University (College Station, TX). He is currently a Senior Software Engineer working on the Sensor Prime run-time and database products at MultiGen-Paradigm.

Virtual Texture: A Large Area Raster Resource for the GPU

Anton Ephanov, Chris Coleman
MultiGen-Paradigm
Richardson, TX

INTRODUCTION

Texture mapping has traditionally played a key role in real-time 3D computer graphics, where it is used as the primary means for adding realism to the scene. Techniques and approaches that utilize texture mapping evolve with innovations in 3D graphics hardware. The fixed OpenGL rendering pipeline is being replaced by the programmable pipeline using vertex and fragment shaders. The programmable graphics pipeline revolutionized the world of real-time 3D graphics by introducing unprecedented flexibility in utilization of graphics resources. Programmable Graphics Processor Units (GPUs) allow techniques which utilize textures as a generic look-up resource, thereby allowing textures to represent non-visual information about the database area, such as spectral data at multiple wave bands, thermal data, normal displacement maps for improved terrain shading, digital elevation maps for the terrain shape, or material-encoded maps for parametric approaches to providing dense organic scene content.

Many of today's advanced rendering techniques require multiple channels of raster data. These techniques can be applied through shaders and multi-texture stages on models and small terrain areas using traditional modeling tools. However, when the area of simulation becomes large, the raster resources must be managed to take maximum advantage of hardware resources to achieve the best quality and performance combination appropriate for the scenario.

Motivation

Texture mapping has always been a very demanding resource requiring significant amounts of main computer memory (RAM) and video (texture) memory. Simulations that involve visualization of large database areas, such as flight simulators, are especially challenging in this respect. For large area visual simulations, the total amount of raster information for a database typically far exceeds available computer and video memory. For instance, a typical database design for a high to medium altitude range flight simulator might have a coverage of 900 by 700 kilometers of geo-specific satellite imagery at 0.8 meters per pixel. The corresponding texture dimensions are on the order of 1125000 x 875000 pixels. At 3 bytes per pixel, covering the entire area with high resolution imagery would require nearly 3 terabytes of data. Further, advanced rendering techniques could require far more than 3 bytes per pixel of data. There is no graphics hardware that can satisfy such exorbitant memory requirements. Therefore, the image generator subsystem faces a substantial data management problem where it has to provide and combine heterogeneous resources (textures containing various data formats) to achieve the desired image quality and real-time performance characteristics. The data management (streaming) aspect of large-area coverage is equally important. The approach should be inherently efficient and scalable to address the challenges of modern combat training, where sometimes only a limited bandwidth is available for on-demand database streaming.

The data paging is essentially a two-stage process that involves paging from disk to RAM first, and moving the data from RAM to video memory second. Recent advances in the development of faster AGP and PCI Express transfer buses make commodity PC hardware capable of the desired data throughput. As a result, producing compelling image quality and a highly dynamic scene content at real-time frame rates using huge textures is a reality, providing that there is a software solution that offers an efficient run-time management of the image resources.

PREVIOUS WORK

Techniques for dealing with huge textures range from subdividing the texture into smaller tiles that can be directly supported by the graphics hardware to offering specialized low-level graphics hardware and high-level system software such as clip-mapping (Tanner, Migdal, and Jones, 1998). Both ends of the solution spectrum have advantages and limitations. The texture tiling approach is cost-effective because it can be implemented on commodity graphics hardware. The approach inherently provides good paging granularity and deterministic image quality. However, it also presents significant challenges. First, the tiling approach requires that geometric primitives must not cross the texture tile boundaries when using the fixed OpenGL pipeline. With a typical maximum hardware-supported texture size of 4096 x 4096 texels, the limitation translates into stringent geometry tessellation requirements that add an extra level of complexity to the database design. Second, the tiling approach complicates the Level Of Detail (LOD) management both at the geometry and the texture levels, where it can produce visually distracting popping artifacts. Third, the approach tends to be sub-optimal from the run-time state management perspective, where rendering a single frame may require a significant number of texture binds, introducing a considerable run-time performance penalty. Finally, the approach requires that the texture tiles are assigned to the geometric primitives at the database design level, therefore implicitly making the overall run-time texture budget dependent on the database design and the LOD management scheme. This aspect of the tiling solution makes the texture budget planning and performance tuning particularly complicated. It also makes it impossible to add new techniques such as bump mapping onto an existing terrain without regenerating the database.

The clip-mapping approach (Tanner, Migdal, and Jones, 1998) has been de facto the preferred solution for large area database visualizations on Silicon Graphics (SGI) workstations. It produces high quality images in real-time from very large textures using relatively little texture memory. The approach greatly relaxes the geometry tessellation limitation of the tiling approach. From the end-user perspective, the major disadvantage of clip-mapping is that it is only available on a selected set of SGI workstations. The inability to utilize this approach on commodity PC hardware makes it inflexible and cost-prohibitive. Clip-mapping has other downsides as well. Fundamental limitations of the hardware can manifest themselves under certain conditions as distracting visual artifacts such as the “jello” and “wobble” effects. These effects appear as tears in the imagery or temporal image shifting due to numerical precision issues and clipping of large polygons. Additionally, clip-mapping can not be combined with other texture extensions (such as SGI detail texture) or modern shader-based texturing techniques, thereby making this solution less attractive from the image quality perspective by today’s standards.

The Virtual Texture (VT) approach presented in this paper falls in between the tiling texture and clip-mapping approaches. Virtual Texture started as a research project to provide the customers the ability to migrate their clip-map databases to the PC platform with a minimal amount of database modifications. Therefore, the Virtual Texture and clip-mapping formulations share many fundamentals. Clip-mapping is patented technology of SGI (Migdal, et al., 1995). Virtual Texture’s emulation of clip-mapping behavior on the PC platform is patented technology of MultiGen-Paradigm (Ephanov, 2000). It should be noted that the patented implementation is limited to the fixed OpenGL pipeline only with a single texture per level of resolution. The ideas in this paper present continuing innovation based on previous work to take full advantage of the programmable nature of modern graphics hardware.

THE VIRTUAL TEXTURE FORMULATION

The main objective of Virtual Texture is to manage large amounts of raster data while providing data to the GPU as a set of textures.

The Design Requirements

1. Virtual Texture must act as though it were a regular texture as much as possible. From the database modeling perspective, one should be able to apply VT onto geometric primitives, where texture coordinates of the vertexes should be assigned the same way as if mapping a regular, albeit huge, texture. Therefore, the texture coordinates are assumed to be within the [0,1] range for the entire area covered by the texture, usually the entire database.
2. It must be possible to use Virtual Texture in combination with advanced rendering techniques often utilizing vertex and fragment shaders.
3. It must be possible to combine multiple Virtual Textures for rendering in a single pass, therefore making Virtual Texture multi-texture friendly. A variety of rendering techniques use multi-texturing effectively for improving the overall image quality. For instance, an advanced shading technique such as DOT3 bump-mapping can be extended to work with two Virtual Textures representing a color map and a normal displacement map respectively.
4. At a minimum, the implementation must only require capabilities that are available in OpenGL 1.2. This requirement makes Virtual Texture cross-platform and cost-effective.
5. The implementation must meet high-performance real-time requirements, such as a stable frame rate of 60Hz or above, single-pass terrain rendering, and a data throughput that is capable of sustaining

fast observer motion at Mach 1 or higher. Minimizing the number of texture applies per frame and throttling of sub-loading are required.

6. The Virtual Texture formulation must be flexible to allow for data to be stored or streamed from many different formats and image file sizes. All of the Virtual Texture concepts discussed in this paper are independent of how the raster information is stored on disk.

Levels of Virtual Texture Resolution

We begin this section with an obvious realization that if the texture is huge, we can't utilize all the image data for rendering at once – at least not at real-time frame rates. This is beyond the memory and the throughput capacities of modern hardware. At the same time, we don't need to display all the imagery at once because we typically see only a limited portion of the total database area. Additionally, the display technology offers resolutions that are either on a par or smaller than the maximum hardware supported texture size. Therefore, in reality, we need to define and visualize a high resolution inset of the entire Virtual Texture. The high resolution inset is a dynamic subset of the image data, whose content changes during run-time depending on the application's visualization needs.

The inset is represented by an array of levels of Virtual Texture resolution (we will also refer to them as Virtual Texture levels or just levels). At runtime, the imagery within each level is defined by the following factors:

1. The resolution of the image data that the level is composed of. By convention, level 0 is considered to have the highest resolution, which is consistent with traditional texture mapping. Each subsequent level represents image data at half the resolution.
2. The location of the paging center. The paging center is the center of the current area of interest. It can be coincident with the observer or positioned at an arbitrary location in the scene. The paging center can be set individually for each level of resolution to reduce paging dependency among the levels to facilitate management of sparse datasets.
3. The level configuration which consists of a square n x n array of level tiles. A level tile is basically a texture, although, in the current implementation, the texture is double-buffered for the reasons that we will discuss later. The dimension n can be chosen individually for each level based on the database design and the image quality requirements. Tiles that comprise a level are of equal square dimension. The tile dimension can be set individually for each level. Such multi-tile formulation allows for significant configuration flexibility, where the user can trade off memory and texture budgets for image quality.

Based on the level definition presented above, let us discuss what we should expect visually. For simplicity, let us assume that all levels are identical in terms of the tile configuration and share the same paging center location. Visually, the levels of resolution should create a set of concentric square enclosures of the image resolutions. The most inner enclosure should present the imagery at the highest resolution, while each next level drops the resolution by half. The concept of levels of resolution is demonstrated on Figure 1, where the levels of resolution are color-coded on the right image for better illustration.

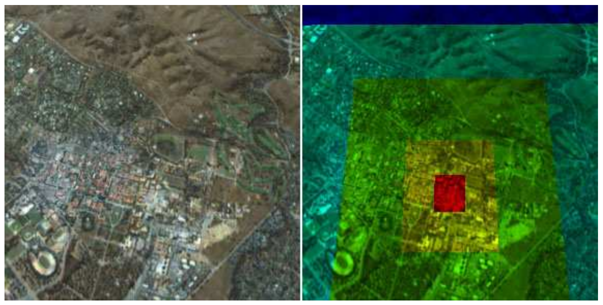

Figure 1. Levels of Resolution of Virtual Texture

The choice to down sample each level by a factor of 2 is motivated by the idea of texture mip-mapping, where a texture together with its mip-map levels can be represented as a resolution pyramid (Neider, Davis, and Woo, 2005). In the mip-map formulation, starting with the highest resolution, each lower level represents the image using half as many texels in each direction. Texture filtering algorithms utilize mip-maps to minimize visually distracting texture aliasing artifacts. From this perspective, Virtual Texture can be represented as a gigantic inverted pyramid, where the base of the pyramid is the image data at the highest resolution, while each next mip-map level represents the image data at half the resolution. Notice also that we don't specify how the image data is stored on disk. The Virtual Texture formulation is decoupled from the specifics of data storage assuming that there is an algorithm to retrieve a given area of resolution. In other words, there is no direct mapping between image files on disk and VT level tiles.

Figure 2 presents a schematic view of Virtual Texture. The top left image represents the entire Virtual Texture mip-map pyramid. Each green line is a level of the mip-map pyramid viewed "edge-on." The image at top right is a visualization of a single mip-map level as viewed from the "top-down." The level of the mip-map pyramid is now a green square. Also shown are the paging center and the high-resolution inset. The image at the bottom left corner shows the high-resolution inset which is configured as a 2x2 array of level tiles. Finally, the bottom right image shows the four level tiles as textures with their own mip-map pyramids.

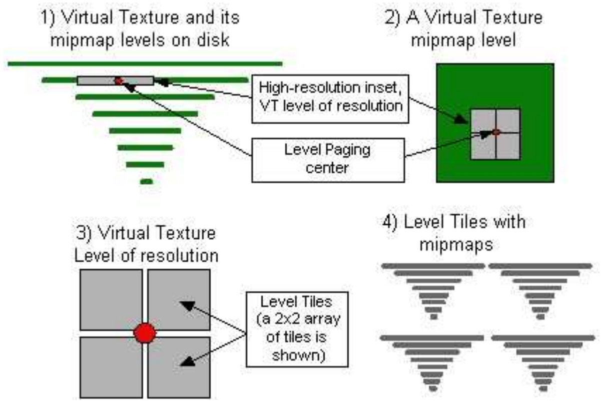
1) Virtual Texture and its mipmap levels on disk
2) A Virtual Texture mipmap level
High-resolution inset, VT level of resolution
Level Paging center
3) Virtual Texture Level of resolution
Level Tiles (a 2x2 array of tiles is shown)
4) Level Tiles with mipmaps

Figure 2. A Schematic View of Virtual Texture

EXAMPLES OF VIRTUAL TEXTURE TECHNIQUES

As it was stated in the previous section, the primary objective of Virtual Texture is to manage large amounts of raster data while providing the data to the GPU as a regular texture or a set of textures. Such an approach makes it possible to utilize Virtual Texture with the majority of GPU-based rendering techniques that already work with regular textures. The ability of Virtual Texture to manage extremely large raster data sets significantly expands the application domain of the GPU techniques in the context of large area databases. In this section, we present examples of GPU techniques and their benefits when applied to large area databases.

Basic Color Imagery

The most basic application of Virtual Texture is to apply geo-specific (i.e. satellite) imagery to the terrain. This application provides functionality that is similar to the clip-mapping approach (Tanner, Migdal, and Jones, 1998). In this context, Virtual Texture is used directly as a color map for texturing. It should be noted that applying extremely large high resolution raster to a relatively small geographical area also falls into this category. It is the amount of raster data that requires Virtual Texture, not the spatial dimensions of the area per se.

Dynamic Terrain Bump Lighting

Another popular application of texturing is simulation of per-pixel lighting using a bump or normal map (Kilgard, 2000). The technique significantly improves image quality by making the terrain shading look more natural and realistic. The classic variant of the bump mapping technique requires two textures – a color map and a bump map. The bump map provides local normal variation on the surface of the object in "tangent space." Tangent space is defined by a set of vectors which make up a basis for each face of the object. This set of vectors is typically pre-computed and stored in a texture coordinate for each vertex of the object. The terrain, being a static object, presents a special simpler case, where the world (scene) space can be used as the tangent space for each vertex, thereby eliminating the need for otherwise complex tangent space computation. Rather than a bump map, which perturbs local normals, a "normal map" which encodes the surface normals in world-space directly can be used.

Both the color and normal maps are very large raster data sets. Therefore, application of bump-mapping to the terrain requires two Virtual Textures. Interestingly enough, the two VTs do not need to represent data at the same resolution. The normal displacement map is typically provided at a much lower resolution than the color map while still producing acceptable image quality. The normal map can be derived by the database generation tools from the available digital elevation data and enhanced using fractal-based techniques.

It should also be noted that time of day generally advances very slowly in simulations. It is possible to compute terrain lighting given information such as a color map and normal map during the loading of Virtual Texture levels so that lighting is computed at load time rather than on the GPU. The advantages of loading the normal map into the GPU are dynamic, continuous time of day without having to re-compute all of the loaded Virtual Texture, and the ability to include the effects of local light sources such as headlights and searchlights with high quality lighting on the illuminated terrain.

Detail Texture

Virtual texture can also be combined with raster data that is not "virtual" in nature. The term "detail texture" typically applies to a variety of techniques that strive to improve image quality in the areas where high-resolution imagery is unavailable by adding an artificial noise pattern - the "detail". Regardless of the implementation details, detail texture approaches are in principle compatible with Virtual Texture due to its formulation. A Virtual Texture essentially behaves as a single texture to the user, leaving other texture units available for advanced rendering techniques.

The implementation of detail texture varies depending on the approach and the graphics hardware. The extension (GL_DETAIL_TEXTURE_2D_SGIS) is used on platforms where it is supported. A multi-texture based approach is used to simulate similar functionality on other platforms. A more sophisticated approach, called "hyper texture," is also possible in this context that utilizes a set of detail textures and combines them progressively with mip-map levels of the base texture to simulate the desired effect (MultiGen-Paradigm, 2006). Additionally, texture blending can be used effectively combine multiple detail textures producing a naturally looking pattern (Nuydens, 2002).

Sensor Simulation

Night vision goggles, infrared cameras, various types of radar, and many other devices require data outside the range of the visible spectrum for a proper simulation. Reflectance or emittance in a particular waveband, thermal information, and radar backscattering properties may be required depending on the type of device simulated. A Virtual Texture is typically built for each type of sensor being simulated.

The material make-up of every pixel in the color Virtual Texture is often used to derive properties for sensor simulation. There are many difficulties inherit in material classification of geo-specific imagery, whether performing classification with a single material per-pixel or multiple materials and mixes per-pixel (Davidson, 2006). Further, material rasters are indexed formats by their very nature, meaning that they cannot be filtered as continuous data. Rather than building a material Virtual Texture, which requires filtering the materials to fill in all levels, the materials data can be converted to in-band data that is analog in nature for a particular sensor. This results in better filtering, access to additional source data besides just materials, more accurate simulation, better image quality, and consistency across image levels.

For infrared simulation, converting materials to static radiance as sensed by an infrared camera is the simplest form of sensor simulation. It results in a static time of day scene. When using static radiance, simulating a different time of day would require building an entirely separate Virtual Texture for each time of day simulated. Alternatively, thermal data and properties such as material reflectance and/or emittance in the waveband of the sensor can be encoded into a Virtual Texture to allow for dynamic time of day for both emitted and reflected energy. MultiGen-Paradigm's proprietary I24 sensor texture format contains indexed thermal data which texture filters, mip-maps, and compresses. Virtual Texture feeds this raster data to the GPU which computes at-aperture radiance for the sensor.

Modern GPUs operate with floating-point precision at extremely high frame-rates, making the PC an ideal platform for large-area sensor simulation. Virtual Texture allows for geo-specific simulation of large area datasets in any wavelength. Further, the GPU makes continuous time of day possible at extremely high frame rates, allowing these techniques to meet real-world training requirements using only a single GPU.

Dynamic Terrain using Elevation Rasters

Elevation data is readily available, and is often used in constructing a polygonal representation of the terrain. This high quality terrain elevation data could instead be converted into a Virtual Texture and the GPU used to modify the positions a static terrain skin to match the elevation data. While the details of these techniques can be found in other papers (Losasso and Hoppe, 2004), the Virtual Texture implementation makes management of the elevation data efficient and configurable.

Horizon Mapping for Terrain Self-Shadowing

In addition to normal maps and elevation maps, horizon maps can be used to compute shadowing of the terrain from the sun or moon. A horizon map stores angles that can be used to determine whether a point on the terrain surface is in shadow for a particular incident azimuthal light direction (Max, 1988). Two solar elevation angles, one in the East azimuthal direction and one to the West, are sufficient to compute solar terrain shadowing at run-time in the GPU. The angles stored in the horizon map represent the east and west "horizons" for each surface point on the terrain.

As with bump mapping, this data can be interpreted at load-time to create a static lighting texture for the terrain, or lighting and shadowing can be computed dynamically on the GPU.

Illumination Maps for Night Simulation

An unlimited number of city lights and light pools can be "baked-in" to a texture for use at night. Raster-based approaches for light pools allow for pre-computed shadowing, colored light sources, and large numbers of individual light pools scattered along roadways, and are far more efficient than geometry-based lighting of a night scene. Geo-specific night scene illumination can be efficiently implemented with a Virtual Texture.

Challenges in Using Color Textures for Other Data

These examples of Virtual Texture techniques show that there are many types of raster data that can be made available to the GPU. Unfortunately, each individual texture is limited to the 4 color components – red, green, blue, and alpha (RGBA). Virtual texture raster datasets with the same coverage and resolution can be combined into a single texture for faster lookup, as long as 4 or fewer components are required. Otherwise, additional texture stages must be used. More data requires more video memory consumption and possibly more texture look-ups per fragment which can impact performance.

Because terrain is often viewed at oblique angles towards the horizon, mip-mapping is almost always required for Virtual Texture. Further, filtering of some type is required to build a Virtual Texture dataset at all. Mip-mapping and texture filtering can be problematic for non-color rasters such as indexed materials and normal maps. One solution, as mentioned in the section on sensor techniques, is to convert from an indexed format into an analog format. Materials data is often converted to some analog data relevant to the sensor simulation. However, even when using analog data in the red, green, blue, and alpha channels, care must be taken in the filtering settings and compression characteristics chosen. For example, hardware supported DDS compression can be problematic for non-color rasters because the compression algorithm makes assumptions, such as putting emphasis on the green channel, that simplify color compression which may not be appropriate for other types of data (MSDN, 2006). There are also hardware supported compression techniques specifically formulated for normal map compression (ATI, 2003). Unfortunately, compression techniques used for color or normals may still not be appropriate for other types of raster data supplied by Virtual Texture to the GPU.

THE INTERNAL ALGORITHMS

In this section, we examine Virtual Texture algorithms. We also discuss limitations of the fixed OpenGL pipeline, ways to resolve them using the programmable pipeline, and possible image quality issues produced by Virtual Texture.

The Texture Coordinate Transformation Algorithm

The texture coordinate transformation algorithm is at the core of the Virtual Texture mathematical engine. The primary objective of the algorithm is to guarantee that texels from a given level of resolution are applied correctly to underlying geometry objects producing consistent visual results across the levels. From now on, we define a geometry object as a set of cohesive geometric primitives, such as triangle strips or quads, that is used as a building block for construction of the database hierarchy. The algorithm works entirely in the texture coordinate space of the database. The choice of the texture coordinate space decouples the Virtual Texture formulation from the world space (i.e. Cartesian XYZ).

The texture coordinate transformation algorithm computes a texture coordinate transformation that is unique for each level tile (and, therefore, for each texture object representing the tile). The transformation is a combination of scaling and translation. Therefore, it is a linear transformation that can be applied via a texture matrix. The latter makes it possible to implement Virtual Texture within the constraints of the fixed OpenGL pipeline (no shaders), although with certain limitations that we will discuss later.

The texture transformation represents a conversion between two texture spaces. The first texture space is the database texture coordinate space. The (u,v) texture coordinates of geometric primitives are given in this space. The coordinates could be used (hypothetically) to sample into Virtual Texture at the highest level of resolution (the top of the pyramid on Figure 2, Diagram 1). The second texture space is associated with the level tile texture (Figure 2, Diagram 4). We assign the capital letters (U,V) to this space. The derivation is identical for both texture coordinates, therefore it is presented only for the u coordinate. We model the transformation as linear with the scaling coefficient *scale* and the translation term *trans* as follows:

$$U = scale * u + trans \qquad (1)$$

The unknown coefficients *scale* and *trans* are computed by solving a system of linear equations that we derive from equation 1 by substituting two pairs of known matching values for the texture coordinate as follows:

$$\begin{aligned} 0.5 &= scale * u_{ctr} + trans \\ 0 &= scale * u_{lc} + trans \end{aligned} \qquad (2)$$

where, the u_{ctr} value is a position of the level tile center. The u_{lc} value is a position of the lower left corner of the tile that is computed as:

$$u_{lc} = u_{ctr} - \dim_{tile} * 2^{(level-1)} / \dim_{vt} \qquad (3)$$

The dim_{tile} variable is the level tile dimension, *level* is the level number that the tile represents, and dim_{vt} is the Virtual Texture dimension. The textures are assumed to be powers of two, therefore we can express their dimensions via the corresponding exponents:

$$u_{lc} = u_{ctr} - 2^{(tile+level-1-vt)} \qquad (4)$$

Solving Equation 2 for the unknowns, we obtain the texture coordinate transformation:

$$\begin{aligned} U &= 2^{(vt-level-tile)} * u + (0.5 - 2^{(vt-level-tile)} * u_{ctr}) \\ V &= 2^{(vt-level-tile)} * v + (0.5 - 2^{(vt-level-tile)} * v_{ctr}) \end{aligned} \qquad (5)$$

Equation 5 can be used directly to form texture coordinate transformation matrix for a tile given its level number *level*, location of the tile center (u_{ctr},,v_{ctr}), the tile texture dimension exponent *tile*, and the Virtual Texture dimension exponent *vt*.

Applying Level Tile Textures to Geometries

In this section, we discuss an algorithm that is the foundation for the Virtual Texture rendering flow. The algorithm determines which level tiles to apply to a geometry object. The selection logic is defined by restrictions imposed by the following two factors: 1) programmability of the graphics pipeline and 2) run-time performance considerations. These two factors also have a tremendous influence on the final image quality produced by Virtual Texture. First, we consider a variation of the algorithm that can be used with the fixed OpenGL pipeline. Next, we extend the algorithm to take advantage of the pipeline programmability via vertex and fragment shaders.

The Level Tile Selection Algorithm

The main objective of the level tile selection algorithm is to assign tile textures to a geometry object for rendering. The output of the algorithm effectively defines the visual result of applying Virtual Texture to the underlying geometry objects that represent the database. The input to the selection algorithm is a texture coordinate bounding box that is defined for each geometry in the database texture coordinate space. The box is aligned with the texture coordinate axes. The level tile selection algorithm utilizes the axis aligned bounding boxes (AABB) for efficient resolution of level tiles as described below.

First, we define AABB in texture coordinate space for each level of resolution of Virtual Texture. The box is a union of the corresponding boxes of the level tiles. For each tile, the center of the box is located at the current tile center, while its extents are defined by the Virtual Texture dimension in texels dim_{vt}, the level tile dimension dim_{level} and the level number *level* as follows:

$$\begin{aligned} u_{\min} &= (u_{ctr} - 0.5 * \dim_{level} * 2^{level}) / \dim_{vt} \\ u_{\max} &= (u_{ctr} + 0.5 * \dim_{level} * 2^{level}) / \dim_{vt} \end{aligned} \qquad (6)$$

Notice that Virtual Texture levels and tiles introduce a hierarchy of axis aligned bounding boxes in texture coordinate space. The hierarchy is also used to improve the run-time efficiency of the level tile selection algorithm by culling level's AABB against geometry's AABB as described below.
For each geometry object, the algorithm iterates over the Virtual Texture levels, starting with the level of highest resolution. For each level, the algorithm compares texture coordinate bounding boxes of the geometry object and the level to cull out levels that do

not overlap with the geometry. The algorithm works differently from this point on, depending on whether the rendering is done using the fixed OpenGL pipeline or shaders.

In the case of the fixed OpenGL pipeline, the algorithm searches for a level with a bounding box that completely covers the geometry box. Once such a level is found, the iteration proceeds over the level tiles, searching for a tile that completely covers the geometry bounding box. If such tile is found, the search stops, the algorithm assigns the tile texture and the corresponding texture matrix (see equation 5) to the geometry object for rendering. Otherwise, the algorithm continues the iteration to the next lower-resolution level in search of a level tile that provides complete coverage of the geometry object. Notice that Virtual Texture always includes a level that covers the entire database with a single tile (texture), although at a very low resolution. Therefore, every geometry object is always guaranteed to be assigned a texture and a texture matrix.

Limitations of the Fixed OpenGL Pipeline

The fixed OpenGL pipeline presents significant limitations in the context of Virtual Texture rendering. The most important limitation is that a tile texture has to cover the underlying geometry object completely in the texture coordinate space to be applied to it. This is primarily due to OpenGL's handling of texture coordinate wrapping. Standard OpenGL pipeline offers several texture wrapping modes. The GL_REPEAT and GL_MIRRORED_REPEAT_ARB are not acceptable because they repeat texels outside the [0,1] range. The non-wrapping modes, such as GL_CLAMP, GL_CLAMP_TO_EDGE and GL_CLAMP_TO_BORDER end up smearing the bordering texels or border color, therefore producing unacceptable results as well. We need a mode where sampling into a texture stops once the extrapolated texture coordinate goes beyond the [0,1] range (it is assumed that the texture transformation as described in section 3.1 has already been applied). Unfortunately, this result is very difficult to achieve efficiently with the fixed OpenGL pipeline. There is no multi-texture mode to cut-in a high-resolution inset over a base texture. As a result, Virtual Texture can not be implemented as a continuously moving set of levels of resolution that utilizes a toroidal texture mapping scheme introduced by clip-mapping (Tanner, Migdal, and Jones, 1998). A tile texture can only be applied to the geometry object when it covers the object completely, which is not the case for the majority of paging center positions if the center moves continuously. One solution to work around the limitation is to make the Virtual Texture paging center hop from geometry to geometry, therefore increasing the chances of geometry coverage by level tiles.

Positioning of the Virtual Texture paging center plays a key role in achieving the desired image quality. Figure 3 illustrates how the center positioning strategy is primarily driven by the limitations of the pipeline.

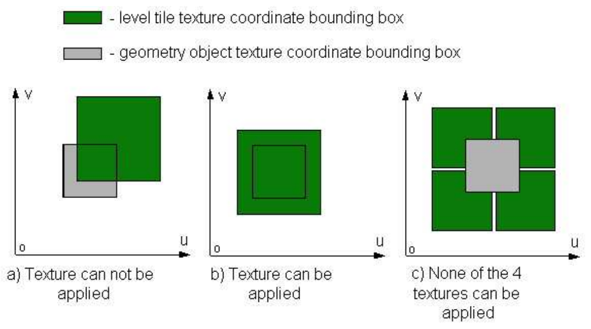
- level tile texture coordinate bounding box
- geometry object texture coordinate bounding box
v
u
0
a) Texture can not be applied
b) Texture can be applied
c) None of the 4 textures can be applied

Figure 3. Limitations of the Fixed Function Pipeline in Texture Coordinate Space

Diagram (a) represents a case where the level tile texture coordinate bounding box does not cover the geometry box completely. The tile texture can not be applied in this case. Diagram (b) demonstrates that aligning the tile center with the geometry makes the texture application possible, therefore emphasizing the importance of the center positioning. Diagram (c) represents a case where a level of resolution is configured as a 2x2 array of tiles. The four texture tiles combined completely cover the geometry bounding box. However, each tile individually does not. Therefore, none of the tiles can be applied to the geometry. A lower resolution level tile which completely covers the geometry would have to be chosen in this context.

Limitations of the fixed OpenGL pipeline visually manifest themselves as noticeable transitions between the levels of resolution that follow the database tessellation because they take place at the geometry borders. A transition from higher to lower level of Virtual Texture resolution when crossing the geometry border can often jump more than one level of resolution. Texels on one side of the geometry border will appear blurrier because they represent a lower level of resolution.

Applying Level Tile Textures using Shaders

Shaders and additional multi-texture stages can be used to improve border conditions between Virtual Texture levels because the programmable pipeline gives us the ability to combine multiple level tiles on a geometry object in a single rendering pass. We can achieve superior image quality, while significantly relaxing the geometry tessellation requirements. The basic idea is to combine multiple tile textures using a fragment shader, where a tile is chosen for rendering if it overlaps the geometry object in the texture coordinate space. The advantage is that geometry does not need to be tessellated, or broken into separate objects to apply high resolution Virtual Texture levels. The disadvantage is that more multi-texture stages are required. The shader combines multiple tile textures for rendering based on the interpolated texture coordinate for each texture stage. Starting with the stage that contains the highest resolution, the shader samples into the texture while masking the result based on the sampling results for the previous stages as follows: texture stages for which the coordinates fall outside the [0,1] range are masked out – this is the desired texture clamping behavior that was missing in the fixed OpenGL pipeline. At the same time, the shader masks lower resolution textures from being applied if and only if a higher resolution texture has already been sampled successfully for the fragment.

The shader case introduces more variation into the level-tile selection algorithm. The key difference is that the algorithm is allowed to associate more than one tile texture with the geometry object. The maximum number of available texture stages is specified by the user within the hardware limitations. The key requirement for the selection algorithm is to guarantee that in the end the geometry object is completely covered (in texture coordinate space) by a set of selected tile textures.

Limitations of Applying Level Tile Textures using Shaders

The shader-based implementation is subject to a limitation by the number of available texture stages (it is assumed that the rendering is to be done in a single pass for performance reasons). This number is hardware-dependent, with eight texturing units being a typical number at the time of this writing. Consider Diagram (c) on Figure 3 again, where the geometry object requires four texture stages for rendering. This is a typical situation for a multi-tile Virtual Texture configuration. As shown, we need four texture stages to render the geometry in a single pass. The number is going to double (or even triple) if we choose to improve the image quality with an advanced rendering technique that requires multiple textures for passing the additional data.

The run-time rendering performance is also a concern. The majority of work is done by the fragment shader. Therefore, the overall rendering performance is dependent on the complexity and efficiency of the shader code, where the complexity increases with the number of utilized texture stages. From this perspective, database designs and Virtual Texture center positioning strategies that facilitate application of fewer texture stages on the fragment shader should be preferred because they produce better run-time performance.

Finally, geometry tessellation presents a rendering performance concern from the perspective of moving the primitives to the pipeline. Many researchers believe that the total number of rendered polygons is less of a performance issue on modern hardware platforms (Losasso and Hoppe, 2004), making many terrain design approaches that strive to minimize the total number of polygons somewhat obsolete. Vertex buffer objects (VBO) offer a very efficient means for providing the vertex data to the pipeline. Proper batching of the primitives becomes the key to fast data transfers, where smaller batch sizes are typically less efficient. However, the tessellation requirements imposed by Virtual Texture oppose batches of large sizes, especially at the high level of resolution. Therefore, one needs to find a compromise between the database design and the vertex transfer approach that would allow for the optimal performance to image quality ratio.

Double Buffering

Each level tile is associated with two texture objects – one for data update and one for visualization. The two textures swap when the data update is complete. The actual image data that is represented by the textures depends on the level number and on the current location of the paging center. The data is very dynamic - it has to be updated as the center moves. The level tile textures come with their own mip-map levels to facilitate texture filtering. But, unlike regular textures, the mip-map levels are not computed at run-time by down sampling the top level. Instead, they are copied directly from the corresponding areas of lower resolution from the Virtual Texture dataset. This is necessary to guarantee cross-level image correlation. It is also cheaper computationally at run-time because the down-sampling algorithm is replaced with a memory copy.

The process of updating a tile texture includes two stages. The first stage updates the mip-map buffers with the new data by paging it from disk to RAM. The second stage, also referred to as texture sub-loading, moves the new data to texture memory. The first stage can be done asynchronously in a separate thread, whereas the second stage needs to be done in the context of the drawing thread because it requires the OpenGL draw context. Double-buffering of textures increases the overall texture budget, which is an important run-time performance consideration.

Texture Sub-loading

Texture sub-loading is a process of moving the image data from RAM into the texture (video) memory. Texture sub-loading is a time consuming process that needs to be controlled to guarantee the frame rate stability. A robust implementation of the texture sub-loading controller is absolutely essential for high-performance applications that utilize Virtual Texture. The actual control law depends on many factors, including the image format, the system bus throughput, the graphics driver implementation, etc. One of the key benefits of compressed texture formats is sub-loading performance.

Numerical Precision Issues

Numerical precision issues typically manifest themselves as distracting visual artifacts, such as imagery "shifts". The graphics pipeline operates using 32-bit floating point precision numbers (the float data type in C/C++) for both texture coordinates and texture matrices. A simple computation shows that it is easy to run out of floating point precision when applying a large-scale Virtual Texture. The C/C++ run-time library defines the number of decimal digits of precision (typically 6) and the "floating point epsilon" value, which is the smallest number such that one plus epsilon is not equal to one (typically on the order of 1.192092896-07F). These constants provide the guidelines for the degree of resolution that one can expect from Virtual Texture. In the Virtual Texture formulation, texture coordinates span the [0,1] range in each direction. Given the numerical limitation of the float type, one can only assign texture coordinates with a minimal difference between any two coordinates of $1/2^{20}$. In reality, a texture coordinate step size much larger than epsilon is required. Therefore, the database can not be tessellated in the texture coordinate space beyond the floating point limitation. Possible solutions to this limitation are to reduce the Virtual Texture size or to introduce a texture coordinate localization scheme, where a stack of double-precision texture matrices can be used to compensate for floating point deficiencies.

Another numerical issue is related to the translation term of the texture transformation matrix. The term is derived from equation 7 as follows:

$$translation = (0.5 - 2^{(vt-level-tile)} * u_{ctr}) \qquad (7)$$

where, u_{ctr} is the location of the paging center in the database texture coordinate space, *level* is the level number, *tile* is the level tile dimension exponent, and *vt* is the Virtual Texture dimension exponent. Since levels of lower resolution are associated with higher level numbers in the Virtual Texture formulation, the overall scaling factor that multiplies the paging center u_{ctr} gets smaller with lower resolution, exposing the floating point limitations. This issue produces a visual "shift" in the lower-resolution imagery when the paging center moves. A possible solution to this problem is to introduce an imaginary grid in texture coordinate space and clamp the paging center to it (recall that each level of resolution can be centered individually). The grid points should be computed such that the lowest decimal digits are equal to one over a power of two, where the exponent value can be chosen empirically or derived from the database specification. Equation 7 will produce less numerical error when the center is clamped to such a grid.

State Sorting for Optimal Rendering Performance

As described in previous sections, the level tile selection algorithm associates tile textures with the geometry objects. The rendering performance can be optimized if the geometry objects are sorted by the tile textures upon completion of the selection algorithm. The additional sorting pass minimizes the total number of texture applies (binds) that is required for rendering the database. In fact, the entire database may end up being rendered with as few texture binds as the number of distinct level tiles that have been chosen for rendering. The number of binds is typically low, even in the presence of shaders that require multiple texture binds per geometry. The ability to minimize the total number of texture binds brings significant run-time performance benefits, making Virtual Texture a very attractive solution from this perspective.

CONCLUSIONS

Virtual Texture is a novel approach to texturing that allows for rendering of large database areas with a texture of exceedingly large dimensions, such as visualization of large-scale geographical areas with geo-specific satellite imagery. The primary difficulty with such visualization is that the amount of image data that is used by the visualization by far exceeds available computer memory and video resources. The key benefit of Virtual Texture is that it allows for efficient visualization with the imagery at real-time frame rates on a commodity PC platform.

Scalability is another feature of this approach, where the user can trade off computer resources for improved image quality. An additional benefit of Virtual Texture is ease of integration with modern visualization techniques, such as multi-texturing, vertex and fragment shaders, and vendor-specific texturing extensions such as detail texture. This feature allows for achieving superior image quality by integrating geo-specific imagery with other resources such as normal displacement maps for improved terrain shading, multiple representations for different sensor spectral bands, thermal data for infrared simulation, and detail texture for enhancing image quality in the areas where high-resolution satellite imagery is unavailable.

REFERENCES

ATI. (2003). *Normal Map Compression*. Retrieved June 16, 2006 from http://www.ati.com/developer/NormalMapCompression.pdf

Davidson, S. (2006). Material Classification Pragmatics: Creating and Evaluating Geo-Specific Material Assignments. *Paper Submittal for I/ITSEC 2006*.

Ephanov, A. (2000). United States Patent 6,924,814. Issued August 2, 2005.

Kilgard, M. (2000). A Practical and Robust Bump-Mapping Technique for Today's GPUs. *Proceedings of Game Developer's Conference 2000, Advanced OpenGL Game Development*.

Losasso F., & Hoppe H. (2004). Geometry clipmaps: Terrain rendering using nested regular grids. *SIGGRAPH '04 Proceedings*, pages 769-776.

Max, N. (1988). Horizon Mapping: Shadows for Bump-Mapped Surfaces. *The Visual Computer, Volume 4*, pages 109–117.

Migdal , et al. (1995). United States Patent 5,760,783. Issued June 2, 1998.

MSDN Article. (2006). *Compressed Texture Resources*. Retrieved June 16, 2006 from http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/Compressed_Texture_Resources.asp

MultiGen-Paradigm. (2006). HyperTexture - a Virtual Texture Technique. *Vega Prime Programmer's Guide*.

Neider J., Davis T., & Woo M. (2005). *OpenGL Programming Guide (5th Edition).* Addison-Wesley.

Nuydens, T. (2002). *Terrain Texturing*. Retrieved June 22, 2006 from http://www.delphi3d.net/articles/viewarticle.php?article=terraintex.htm

Tanner, C., Migdal, C., & Jones, M. (1998). The Clipmap: A Virtual Mipmap. *Computer Graphics, SIGGRAPH '98 Proceedings*, pages 151-158.