4 Game Technology Evolution

This chapter provides an overview of the technology employed in 3D computer games over the last decade. Technology in this context means both software technology, e.g., the algorithms and techniques employed, as well as hardware technology, which has become important with the advent of powerful low-cost consumer graphics hardware in 1996.

Section 4.1 covers seminal computer games, and section 4.2 is devoted to 3D hardware accelerators.

4.1 Seminal 3D Computer Games

In this section, we illustrate the evolution of 3D computer games over the last eight years by looking at several seminal games of that period. Prior to 1992, computer games were either not three-dimensional at all, or used simple wireframe rendering, or maybe flat-shaded polygons. In some cases, 3D information was pre-rendered or drawn into bitmaps, and used at run-time to produce a three-dimensional impression, although there was no actual 3D rendering code used in the game itself.

Ultima Underworld (Looking Glass Technologies, 1992)

In Ultima Underworld, a role-playing game set in the famous Ultima universe created by Origin Systems, players were able to walk around in a fully texture-mapped [Hec89, Hec91] 3D world for the very first time. Most role-playing games prior to Ultima Underworld were not three-dimensional at all, e.g., using a top-down or isometric perspective and tile-based 2D graphics like the earlier Ultima games. One could not walk around in these worlds in the first-person perspective, so the feeling of "actually being there" was rather limited.

Another earlier approach was to combine hand-drawn graphics tiles with a very restricted first-person view, which was originally made popular in the 1980s by the game Dungeon Master, by FTL. However, the world in Dungeon Master was not actually 3D; player positions were constrained to a predefined grid and the only allowed rotation of the view was in steps of 90 degree angles, so one could only look straight ahead, straight to the left or right, or to the back. This approach creates a limited three-dimensional impression without using any actual 3D calculations at run-time.

The world in Ultima Underworld, however, contained basically no technological viewpoint restrictions and was – at least to a certain extent – fully 3D. The player position was not constrained to a simple grid anymore, players were able to seamlessly walk over actual texture-mapped floor polygons. Rotations about the principal axes were also allowed, most notably looking to the left or right with arbitrary rotation angles, and looking up or down with proper perspective fore-shortening of polygons. This is especially remarkable since it wasn't until several years later that most 3D games were using an actual rotation about the horizontal axis for this purpose, instead of not allowing the player to look up or down at all, or faking the rotation by using a shear transformation.

However, the flexibility allowed for by Ultima Underworld's 3D engine had its price in terms of performance. Since the whole world was texture-mapped and polygons could be viewed at arbitrary oblique angles, the texture-mapper was simply not able to texture the entire screen in real-time, at least not on most consumer hardware available in 1992. For this reason, the game used a smaller window for rendering the 3D view of the world, and used the remaining area of the screen for user interface elements like icons, the player character's inventory, an area for text output, and the like.

For a role-playing game like Ultima Underworld its rather slow speed was no problem, however, since it wasn't a fast-paced action game after all, emphasizing game play and design much more than many 3D action games of the following years.

Characters and objects in Ultima Underworld were not rendered as actual 3D objects, but as simple two-dimensional sprites (billboards) instead. That is, they consisted of bitmaps that were scaled in size according to the viewing distance, and animation was done using a sequence of pre-drawn frames, like in traditional 2D animation.

See figure 4.1 for a screenshot of Ultima Underworld.

Figure 4.1: Ultima Underworld (1992)

Wolfenstein 3D (id Software, 1992)

Shortly after Ultima Underworld, a small game developer named id Software released what would eventually found a new computer game genre: the first-person shooter, commonly abbreviated as FPS. This game, Wolfenstein 3D, emphasized fast-paced action above all, where the player was able to run around in first-person perspective with high frame rates, and the goal of the game was primarily to shoot everything in sight. However, this simple formula was easy to grasp and the fast action game play combined with equally fast frame rates contributed to a remarkable success.

The most important contribution of Wolfenstein 3D, however, was that it was to become the prototype for graphically and technologically much more sophisticated first-person shooters in the following years, like Doom and Quake. So, the questionable game content notwithstanding, Wolfenstein had a much higher impact on games in the following years in terms of technology than Ultima Underworld.

In contrast to Ultima Underworld, the game world in Wolfenstein was highly restricted. The player had only three degrees of freedom (DOF); two degrees for translation, and one degree for rotation. That is, movement was constrained to a single plane and the view point could only be rotated about the vertical axis. So, it was possible to look to the left and right with arbitrary viewing angles, but nothing else.

In order to avoid the negative performance impact of texturing the entire screen, only wall polygons were texture-mapped. The floors and ceilings were simply filled with a solid color.

See figure 4.2 for a screenshot of Wolfenstein 3D.

Figure 4.2: Wolfenstein 3D (1992)

In addition to texturing only the walls, the placement of these walls was restricted in a way that all of them were placed in 90 degree angles to each other, and all walls were of the same height. This restriction of the world geometry, combined with the restriction on the allowed motion of the player, also made use of an extremely simplified texture-mapper possible. Walls could be rendered as a series of pixel columns, where the depth coordinate was constant for each column. So, the perspective correction had to be performed just once for each column, leading to an extremely optimized (and equally restricted) texture-mapper.

The visibility determination in a world like Wolfenstein's also profited tremendously from the simplicity of the game world. The game used a simple ray-casting algorithm, where a single ray was cast to determine the entire visibility (and horizontal texture coordinate) of an entire screen column. Since all walls were of the same height, there was always exactly one wall visible per screen column, or none at all. Thus, as soon as the cast ray hit the nearest wall, the visibility determination for the corresponding column was done.

Similar to Ultima Underworld, all characters and objects in Wolfenstein 3D were two-dimensional sprites.

Doom (id Software, 1993)

When Doom first appeared in 1993, it managed to lift many of the restrictions that made its direct predecessor Wolfenstein 3D a very simple and tremendously restricted game. It is still one of the most successful computer games of all time.

Doom combined fast game play and high frame rates with a fully texture-mapped world like in Ultima Underworld, although the world geometry was still much more constrained. The basic premise was similar to Wolfenstein, but the complexity of Doom's levels was truly remarkable for a game running on consumer hardware at the time.

Doom's levels were represented as a single large 2D BSP tree, which was most of all used for visibility determination purposes. Levels could be constructed using some kind of two-dimensional floor plan, i.e., a top-down view, where each line segment corresponds to a wall and is displayed as a polygon in 3D by the game engine at run-time. Of course, this underlying 2D nature of Doom's levels led to a lot of necessary restrictions like "no rooms above rooms," where it was impossible to model multi-storied buildings or anything else where there are two floor or ceiling heights at the same position of a top-down view. Sometimes, such a combination of 2D geometry actually used to define some sort of restricted 3D geometry via lofting or extrusion is called 2.5D.

Doom was able to texture-map the entire screen at really high frame rates by once again cleverly exploiting the combination of a constrained world geometry with restrictions on the allowed rotation of the view point. Since all wall polygons were extruded orthogonally to the floor from the level representation at run-time, all visible polygons were either parallel to the floor plane, or orthogonal to it. Combined with the fact that view point rotation was only possible about the vertical axis, this yields the observation that the depth coordinate is always constant for an entire horizontal or vertical polygon span; for floor/ceiling and wall polygons, respectively. Sometimes, such an approach to texture-mapping is called "constant z texture-mapping."

See figure 4.3 for a screenshot of Doom.

Figure 4.3: Doom (1993)

When using a BSP tree for visibility determination there are two basic approaches in which order the tree can be traversed in order to obtain correct visibility. The simplest is back-to-front rendering, in which the polygons are simply drawn from farthest to closest, in essence using a painter's algorithm [Fol90]. Since the BSP tree yields a correct visibility order for an arbitrary viewpoint, it is easy to let nearer polygons overdraw polygons farther away and thus obtain a correct image.

The problem with back-to-front rendering, however, is that it draws a lot of unnecessary polygons, since it starts with the polygons that are most likely to not be visible at all. Most polygons of a level are invisible most of the time, because they are overdrawn by nearer polygons completely overlapping them.

A much better, but also more sophisticated, approach is to use front-to-back rendering instead. With this order of traversal, drawing starts with the nearest polygon and progresses to the farthest polygon. The major difference is that as soon as the entire screen has been filled by polygons drawn up to a certain position in the BSP tree, further traversal is completely unnecessary, since all visible polygons have already been drawn at that time.

Thus, Doom used front-to-back traversal of the 2D BSP tree representing the entire level, and kept track of covered pixel spans – which was necessary to prevent occluded parts of polygons farther away from overdrawing visible parts of polygons nearer to the view point – and stopped traversal as soon as it knew that the entire screen had been covered. This tracking of already covered screen areas/spans was much simpler in Doom's 2.5D case than it would have been in the general 3D case.

Doom's use of a BSP tree for level representation had the consequence that dynamically moving geometry was not really possible. Doors, for instance, were actually modeled as very small rooms, where the ceiling height was modified on-the-fly. It wouldn't have been possible to model swinging doors, or the like. This was due to the fact that BSP trees are not really suited to handling dynamic geometry, especially not in real-time. Also, the building process of a BSP tree – the so-called BSP tree compilation – is very time-consuming, so there was a significant delay between designing a level and actually being able to try it out in the game.

Similarly to Ultima Underworld and Wolfenstein 3D, Doom still used simple sprites for characters and objects.

Another important aspect of Doom, albeit not graphics-related, was that it supported Novell's IPX protocol for network game play on local area networks. Thus, it became the first multiplayer game with sophisticated 3D graphics that was played by a staggering number of people all over the world.

Another trend that started with Doom and had an extremely wide impact on computer games being developed after it, was that it was also highly user-extensible. Players could easily substitute graphics, sound effects, and the like, and even create their own levels with a wide variety of level editors that were developed by community members and made available for free in most cases. Today, many games come with a level editor out of the box and a thriving community is working on new content, as well as game modifications using source code released by the game developers themselves.

Descent (Parallax Software, 1994)

In 1994, a previously unknown company called Parallax Software released a game that featured a fully 3D, 360-degree, six degrees of freedom action game – Descent.

Descent was the first computer game that was able to render arbitrarily oriented polygons with perspective correct texture mapping at frame rates high enough for an action game. Most astoundingly, Descent featured a true three-dimensional world where the player could navigate a small spaceship through texture-mapped underground mines. It was a first-person game, but the fact that the player was controlling a spacecraft, instead of a character walking on foot, allowed all axes to be basically equal.

The visibility determination in Descent was not done with a variant of the BSP tree approach, but instead with a subdivision of the entire world into convex cells interconnected via portals. A cell in Descent consisted of a solid "six-face," a cube that could be distorted arbitrarily, as long as it stayed a convex solid. Thus, each cell had exactly six walls, which could be either solid (mapped with a texture and impassable), or a portal, where the player could see through and fly to the neighboring cell.

See figure 4.4 for a screenshot of Descent.

Figure 4.4: Descent (1994)

In order for a cell structure connected via portals to be traversed effectively, some kind of adjacency graph is usually used. Each node in the graph corresponds to a cell of the subdivision, and each edge describes a connection between two adjacent portals. Using this information it is very easy to start in a specific cell and jump from cell to cell by passing through interconnecting portals.

In such a system, traversal starts with the cell that contains the view point and some kind of depth-first algorithm is employed to recursively render the entire world. After rendering all solid walls of the cell containing the view point, all portals of this cell are touched and for each one of them everything is rendered that is visible through that particular portal.

Cells as building blocks of a world can also be used to store additional attributes, like lighting information, or whether that area (volume) is poisonous, has different gravity, etc. In Descent, the level designers could assign an ambient light intensity for each of the eight cell vertices. At run-time, vertices could be lit dynamically by moving light-sources like lasers, and the like, combining the dynamic lighting value with the static ambient light intensity on-the-fly. Lit polygons were rendered by applying Gouraud-shading [Rog98] to the textures of a cell's walls.

One of the major properties of Descent's world representation was that the cell subdivision was not built during a post-process after designing a level, as would be done in a BSP-based structure. Instead, the cell subdivision was an intrinsic part of the representation and the design process itself. The data structure used at run-time was basically constructed in its entirety while a level was being built. That is, since the level designers used the exact same geometric building blocks the engine itself used, and designated cell walls manually as either being solid (by assigning a texture), or being a portal (by not assigning a texture), no post-process subdivision and portal-extraction was necessary. The adjacency information of cells was also manually created while constructing a level, since new cells had to be explicitly attached to already existing cells when it should be possible to pass from one cell to another. There were also no problems with t-junctions [McR00], say, an edge of one cell touching a face of another cell in its middle. This was achieved by requiring that neighboring cells always share exactly four vertices, so the connection would be created from one face to another face, over the entire surface of each of these faces.

Many of these constraints on the design process could be viewed as a liability, but in the case of Descent it was very much tailored to the type of game. With such a "distorted-cube"-based approach to level-building it was actually very easy to build the interior of mines, mostly consisting of rather long tunnels with many twists and bends. All in all, Descent was a unique combination of game-design and technology, where each of these crucial parts was ideally tailored to its counterpart.

The characters in Descent, mostly hostile robots, were also not restricted to two-dimensional sprites anymore, like in nearly all texture-mapped 3D games that preceded it. Instead, the robots prowling Descent's sprawling underground mines consisted of texture-mapped and diffusely lit polygons. In addition to these polygonal 3D objects, however, Descent still used billboards for some objects, like power-ups floating in its tunnels, and weapons projectiles, for instance.

Descent used also many other clever restrictions in order to achieve its high performance. For example, all cell walls were textured using textures of a single size, namely 64x64. This allowed the use of texture-mapping routines hard-wired for this size, and made a lot of other administrational tasks with respect to textures easier. Texture-mapping was done using a linear interpolation of texture-coordinates for sub-spans of the horizontal spans of each polygon [Wol90], usually performing the perspective division only every 32 pixels. There were also a lot of highly specific assembly routines for texture-mapping, like one texture-mapper using a texture without any lighting, one for texture-mapping with lookup into a global ambient light or fading table, one for texturing and Gouraud-shading at the same time, and so on.

Descent still used fixed point and integer arithmetic for all of its geometrical computations like transformations, as did practically all of its predecessors.

Quake (id Software, 1996)

Quake was the first first-person shooter that used true three-dimensional environments. It was also able to render geometry that was several orders of magnitude more complex than what Descent had been able to use. In order to achieve this, its developers had to use quite a lot of 3D graphics techniques that had previously never been used in a computer game.

See figure 4.5 for a screenshot of Quake.

Figure 4.5: Quake (1996)

The basic level structure in Quake used one huge 3D BSP tree representing the entire level. In some ways this could be viewed as being simply an extension from using a 2D BSP tree in Doom for 2.5D geometry, to using the 3D variant for arbitrary 3D geometry. In reality, however, 3D BSP trees are much more complicated to use for real-time rendering of complex environments than their 2D counterparts. This is true for many aspects of the transition from a rather restricted 2D-like environment to full 3D.

First, all polygons can basically be oriented arbitrarily, which mandates a texture-mapper that is able to map such polygons in real-time. Quake achieved this by performing the necessary perspective division only every 16 pixels and linearly interpolating in between. The difference between this approximation and the true perspective hyperbola was virtually not noticeable, since the length of these subspans was chosen accordingly.

Prior to Quake, not many games requiring extremely high frame rates were using floating point arithmetic instead of fixed point arithmetic, since the FPU of the processors at the time was simply not fast enough. In 1996, however, it became feasible to use floating point operations heavily even in computer games, and Quake did so, all the way down to the subspan-level of its texture-mapper. The perspective division for texture mapping was still done using the FPU, linear interpolation and pixel-arithmetic was then done in fixed point or integer arithmetic. Nearly all higher-level operations like transformation, clipping, projection, were done entirely in floating point.

There are two things to keep in mind with respect to this and the primary target platform of Quake, the Intel Pentium processor. First, conversion from floating point to integer (or fixed point) values, was always (and still is) an inherently slow operation. Thus, actual pixel arithmetic was always faster using integers, even if the performance of the Pentium processor's FPU was comparable or faster than integer operations. After all, in order for a frame buffer or texture to be accessed one always needs integer coordinates (offsets). Second, there was a very specific reason for this exact point of transition from floating point to integer arithmetic. The Pentium processor was able to interleave floating point and integer operations, since its FPU and integer unit are separate and able to operate in parallel. Actually, Quake's texture-mapper was able to interleave the perspective division with the interpolation and rendering of the actual pixels for a 16-pixel subspan, getting the much-feared division essentially for free.

Second, although a 3D BSP tree yields a correct visibility order for polygons, there are a lot of polygons that are contained in the view frustum but still not visible, because they are occluded by nearer polygons. A very powerful solution to this problem is the notion of potentially visible sets (PVS). With potentially visible sets, for each cell in the world (each leaf of the BSP tree), a set of other – potentially visible – cells is computed in a preprocess and stored along with the level data. At run-time, the set of potentially visible cells is retrieved for the cell containing the viewpoint and only polygons belonging to those cells are even considered for rendering. The PVS information is a so-called conservative estimate for visibility. That is, polygons not actually visible may be contained in such a set, but at least all visible polygons are required to be contained. Naturally, the tighter the visibility estimate is, the fewer unnecessary polygons will be rendered at run-time.

Another challenging problem with using a 3D BSP tree is how to render dynamically moving objects. The tree can be used to obtain a correct visibility order for the static world polygons, but polygons that are not part of the tree structure cannot be handled easily. One approach is to clip dynamic polygons into the leafs (convex cells) of the BSP tree, rendering them when the cell is rendered. This is a rather time-consuming process, however. Therefore, Quake uses a different approach to handle dynamically moving objects, like enemy characters.

Before we discuss how moving objects are handled in Quake, it is important to realize that standard z-buffering was basically never used for visibility determination in fast, software-rendered computer games. All the more complicated approaches like using BSP trees, portals, and the like, are – apart from other very important uses like collision detection, and higher level occlusion culling – in a way a faster but more complicated approach to visibility determination than the simple z-buffer approach. However, z-buffering with the necessary comparison per pixel, interpolation of z values, conditional stores into the z-buffer, etc. was in almost all cases too slow to be used in software rendering. (Note that this changed significantly with the introduction of hardware accelerators, which made the use of simple z-buffering for visibility detection at the pixel level feasible for the first time.)

Quake, however, used a clever variant of z-buffering to combine moving objects with the BSP-rendered world. While rendering the static world, filling the entire screen, a z-buffer was filled at the same time. From a performance point of view this is vastly different from a full z-buffering approach, since each entry in the z-buffer gets written exactly once, and there are no z value comparisons and z-buffer reads necessary. This is possible since the depth information is not used for rendering the world. After the entire screen has been filled with world polygons, the z-buffer contains a valid z-footprint for the entire scene. The relatively small number of pixels of moving objects' polygons can then be rendered using standard z-buffering.

Quake also introduced an entirely new approach to lighting a level, using so-called light maps. Static lighting information was precomputed for patches covering all polygons of the level and stored in special texture maps, at a much lower resolution than the texture maps themselves, say, a light map texel every 32 texture map texels. In the original Quake simple light casting was used for calculating light maps, however, a full radiosity [Ash94] preprocess was employed later on.

In every computer game another important aspect is the approach to building a level. As has also become apparent in the section on Descent, level construction and representation is a very important aspect of every rendering engine. Quake introduced the CSG (constructive solid geometry) modeling paradigm to computer games. When building a Quake level, the designer is always using solid building blocks and combining them with the already constructed part of the level via boolean set operations (like in CSG trees). This is a very powerful modeling paradigm, easy to use, and among other useful properties guarantees that the entire level will be solid in the end, since single polygons are never used during construction.

Quake (more specifically, QuakeWorld) also extended the multiplayer game play of Doom to full-blown Internet game play, which was played by thousands of people all over the world, at the same time.

GLQuake (id Software, 1996)

Quake used a proprietary software-renderer that had been developed from scratch, like practically all computer games at that and earlier times. Quite soon after the original Quake had been released, however, id Software released a Quake executable that was able to take advantage of the hardware acceleration offered by boards using the Voodoo Graphics accelerator by 3dfx Interactive. To distinguish this version of Quake from the standard version it was called GLQuake, since it used the OpenGL API that had previously been almost exclusively used on expensive high-end graphics workstations.

The timing was almost perfect, and GLQuake together with Voodoo Graphics accelerators achieved the breakthrough for consumer hardware acceleration in 1996. Previous attempts of hardware accelerators to take a hold in the consumer marketplace failed, most of all due to mediocre performance and the lack of a killer application, i.e. a top-notch computer game. GLQuake, however, became that killer application. Software-rendered Quake had severe performance problems in resolutions significantly higher than 320x200, say, 640x480, on all but the fastest computers of its time. This was mostly a problem of the lowest level of the graphics pipeline, i.e. the rasterizer – turning polygons into actual pixels and mapping them with a texture. The Voodoo Graphics accelerator was perfect for this kind of work and relieved the host CPU of the burden of performing a perspective division for each pixel (or every n-th pixel) which was the most important factor preventing 3D computer games from going to resolutions of 640x480 and higher.

See figure 4.6 for a direct comparison of software-rendered Quake and hardware-rendered GLQuake, for the same frame rate.

Figure 4.6: Quake vs. GLQuake (1996)

In addition to taking over the main load of rasterization, the Voodoo Graphics accelerators were also able to use bilinear filtering instead of point-sampling textures, which led to a much higher image quality. MIP-mapping could also be performed in real-time for each pixel individually, instead of just choosing an approximately suitable MIP-map level for an entire polygon, as software-rendered Quake had done for performance reasons.

In contrast to software-rendered Quake where light maps were combined with the base texture maps before actually using them for texturing using a sophisticated surface caching system, GLQuake rendered the light maps directly as a second pass, using alpha-blending. If the hardware supported it (Voodoo 2, Riva TNT, ...) base textures and light maps could even be rendered in a single pass, using single-pass multi-texturing.

Quake 3 Arena (id Software, 1999)

The rendering engine of Quake 3 Arena is without a doubt the current state of the art in real-time 3D rendering technology in computer games, and consumer applications in general. It is also the first high-profile computer game that requires a 3D hardware accelerator – software-rendering is not even optional anymore.

Quake 3 still uses much of the underlying technology introduced with the original Quake for the first time. This is largely due to the fact that it is the same type of game, at least from a graphics technology point of view. Regarding game play, Quake 3 is no typical single-player game with a story to guide the player along, but is instead geared entirely towards multiplayer Internet gaming. Single-player is only possible insofar as the multiplayer game can be played against computer-controlled bots.

The entire world is still represented using a single, huge 3D BSP tree, and potentially visible set (PVS) information is the most important approach to occlusion culling. For the first time in a game, Quake 3 uses curved surfaces as a fundamental building block in addition to standard polygonal solid blocks.

In contrast to most applications offering curved surfaces, Quake 3 employs quadratic bézier patches with 3x3 control points per patch, where usually cubic patches are used (needing 4x4 control points). Quadratic patches are of course faster to evaluate than their cubic counterparts, but the developers of Quake 3 have stated that one of the main reasons for using them is also ease of use for the level designers, since it is easier to work with a smaller number of control points. Naturally, this also conserves memory. Note that these quadratic patches are of course converted to triangles before they are submitted to the hardware accelerator for actual rendering.

Quake 3 Arena is a very good example for the current trend of not only going into the direction of using ever more polygons to provide smoother surfaces as well as more surface detail, but of aggressively going into the direction of higher rendering quality, with the use of multi-pass rendering.

The appearance of surfaces is not anymore mostly described by a single texture, a light map, and maybe a couple of other attributes, but is defined by a very flexible and general shading description – a so-called shader. Quake 3's shaders are a simple shading language targeted at real-time rendering. Where in a full-blown shading language like Pixar's RenderMan [Ups90], used for feature films, the shader (a function written in a programming language) is most of the time evaluated for each pixel, the textual definitions of Quake 3's shaders are parsed at loading time, converted to a form suitable to real-time rendering, and then used to control the graphics hardware at run-time.

See figure 4.7 for a screenshot of Quake 3.

Figure 4.7: Quake 3 Arena (1999)

Quake 3's shaders are most of all a very flexible way to control how a surface should be rendered using multiple passes of maybe different kinds. Say, one pass for the light map, four passes for an animated base texture combining four textures on-the-fly, one pass for a reflected image of the surrounding environment, and one pass for volumetric fogging. The shading language can also be used for all kinds of animations, not only texture animations, where several textures can be transformed and composited in real-time, but also color animations, animation of the opacity value, and even vertex animations.

In principle, shaders do not really offer anything that would not have been possible with custom C code prior to their introduction in Quake 3. However, they represent a tremendous increase in flexibility and give much more power and freedom to the artists and level designers. Without shaders, an artist would always have to find a programmer to implement a specific special effect. If a certain effect can be achieved with a shader, however, the artists can immediately do this themselves. Of course, a simple but powerful custom-language for shading specification is much easier to use for everyone, including the programmers, so they open up a much wider field for experimentation with high-quality surface rendering.

4.2 Consumer 3D Hardware

In this section, we illustrate the evolution of consumer 3D hardware accelerators over the last four years, starting with the seminal Voodoo Graphics chip introduced by 3dfx Interactive in 1996.

Voodoo Graphics (3dfx Interactive, 1996)

As already touched upon in the section about GLQuake, the Voodoo Graphics accelerator was the first consumer 3D hardware to actually get used by many computer games and achieve a breakthrough for 3D hardware acceleration in the consumer marketplace in general. One of the major properties of this accelerator that made this possible in 1996 was that boards employing a Voodoo Graphics chipset offered no support for 2D graphics at all. They were exclusive 3D boards and could only be used in conjunction with conventional 2D graphics cards. The video signal of the 2D board was either passed through to the display monitor, or not used at all, and the video signal of the 3D board sent to the monitor instead. Of course, in such a configuration it was impossible to render into a window on the 2D desktop. But since computer games are normally played in fullscreen mode anyway, this was no big problem. It restricted use of these accelerators largely to games, however. Graphics applications like 3D modeling programs, programs for scientific visualization, and the like, were usually not able to take advantage of these early consumer 3D hardware accelerators.

The Voodoo Graphics featured dedicated frame buffer and texture memory, as opposed to a unified memory architecture where frame buffer and texture memory share the same RAM. Thus, it was not possible to increase the amount of texture memory when the frame buffer memory was not used in its entirety. This was no real restriction at the time, though, since the only two screen resolutions supported by the first cards featuring a Voodoo Graphics chipset were 512x384 and 640x480. These boards had 2MB frame buffer, and 2MB texture memory. Dedicated memory for textures has a major performance advantage, since the memory bandwidth to and from texture RAM is exclusively available for texture accesses. In unified memory configurations, on the other hand, the bandwidth needs to be shared between frame buffer and texture traffic. This can be alleviated using a more sophisticated scheme for RAM access, however, in 1996 it was a very sensible decision performance-wise. Naturally, if the texture memory and frame buffer are not shared, it is not possible to render directly into a texture. That is, for rendering into a texture there is no way around reading back the frame buffer and copying it into texture memory to achieve the desired effect. Computer games in 1996, and this still holds true for most of today’s games, did not use algorithms requiring to render into a texture each frame (e.g., dynamic creation of environment maps). So, this was practically no restriction at all at the time.

With respect to color and depth buffer precision, the Voodoo Graphics offered a 16-bit color buffer, and a 16-bit depth buffer. In 1996, this was indeed a tremendous step ahead, since practically all 3D games were using 8-bit color indexes in conjunction with a 256-entry palette, prior to the widespread use of hardware 3D accelerators. In 3D applications, the issue of color resolution doesn’t only pertain to the resolution of the colors as they are stored in the color buffer, but also to the precision of colors contained in texture maps. Usually, the maximum color resolution of the color buffer and the texture data is the same. In the case of palettized 8-bit games this meant only using textures containing 8-bit indexes into a global palette. The Voodoo Graphics supported a number of texture map formats and also featured an on-board hardware palette for use of palettized textures. The highest supported color resolution was 16-bits per texel, in 565 or 1555 configurations, respectively. This, together with the elimination of a global palette shared by the entire screen, led to tremendously increased image quality. However, the first games exploiting hardware acceleration on Voodoo Graphics boards still retained nearly all the restrictions that came from using a global palette in the software-renderer, for the simple reason that this issue is very central to the design of a graphics engine and the artwork used and couldn’t be changed very easily or quickly. So, it took quite some time until most games really started exploiting the capabilities offered by then-recent graphics accelerators.

Textures were supported up to a resolution of 256x256 with a couple of additional restrictions. The aspect ratio could not exceed 1:8 or 8:1, respectively, so it was not possible to use a 256x16 texture, for instance.

Furthermore, texture widths and heights always had to be a power of two. This makes tiling textures (wrapping them around) a lot easier, since the wrap-around can be achieved with simple bit masking, instead of using a modulo operation. Constraints like this only became real restrictions a couple of years later, say in 1999, when the first games without any software-rendering started to appear. Previously, most constraints had been present due to the use of a software-renderer anyway.

A very important reason why the Voodoo Graphics was able to achieve the breakthrough for consumer 3D accelerators, where all earlier attempts had failed, was that nearly all of it features could be used without any performance hit at all. That is, practically all of the flashy new features could be turned on simultaneously without decreasing the frame rate. Earlier accelerators were generally much slower and also got, say, four times slower as soon as depth buffering was turned on. This was not the case on the Voodoo Graphics, and so most of these features got actually used.

Voodoo 2 (3dfx Interactive, 1998)

The Voodoo 2 was the logical successor to the original Voodoo Graphics chipset, offering a lot of improvements in detail, but most of all offering tremendously increased performance.

The most important feature introduced with the Voodoo 2 was single-pass multi-texturing. By offering two texture mapping units (TMUs), the Voodoo 2 was able to texture each pixel with two textures at the same time. This feature was mostly used for light-mapping, made popular with the original GLQuake in 1996, and henceforth used by a very high number of games, many of them using the licensed Quake engine.

With single-pass multi-texturing it was also possible to use trilinear filtering without a frame rate-hit, where odd MIP-map levels had to be accessed by one TMU and even levels by the other TMU, using blending between these two levels for going from bilinearly filtered textures to trilinear filtering with MIP mapping.

The standard configuration of Voodoo 2 boards was 4MB frame buffer memory and two times (for each TMU) 2MB or 4MB texture memory, yielding 8MB and 12MB video memory configurations. This led to an increased screen resolution of 800x600 and a lot more textures being able to be resident in texture memory at the same time.

The Voodoo 2 also offered a couple of additional features like enhanced dithering to 16-bit colors (color and depth buffer precision was still 16-bits), and the possibility for SLI (scan-line interleaving) configurations, where two boards could be combined for effectively doubling the fill-rate, one board rendering even scan-lines, the other board rendering odd scan-lines. But SLI was a pure brute-force approach, requiring twice the texture memory without any real benefit (identical textures had to be resident on each board), and also using two PCI slots (in addition to the one slot needed by the 2D accelerator anyway).

Riva TNT (NVIDIA Corporation, 1998)

In 1998, NVIDIA was the first competitor of 3dfx that was able to come close in terms of performance, and even surpass the line of Voodoo Graphics accelerators in terms of features, with its Riva TNT graphics accelerator. The Riva TNT was the first consumer graphics chip that was able to offer high-performance, high-quality rendering with OpenGL.

The TNT, which stands for twin-texel, was a single-pass multi-texturing architecture offering a 32-bit color buffer, a 24-bit depth buffer, and even an 8-bit stencil buffer. It used a unified memory architecture, and most TNT boards featured 16MB memory, allowing for resolutions of up to 1280x1024. In contrast to the Voodoo Graphics, these boards were combined 2D and 3D graphics boards, so they required only a single PCI or AGP slot for a full 2D and 3D acceleration solution. More importantly, they allowed hardware-accelerated rendering into a window on the normal 2D desktop. This, in combination with support of a fully compliant and quite robust implementation of OpenGL 1.1, brought the breakthrough for consumer hardware acceleration in other applications than 3D games. The TNT also offered maximum texture sizes of up to 2048x2048.

In retrospect, the two main reasons why OpenGL was able to establish itself as a feasible high-performance API in the game development community were probably Quake (GLQuake, Quake 2, ...) on the one hand, and NVIDIA's TNT accelerator on the other. GLQuake was the first high-profile game using OpenGL instead of Glide or Direct3D. The Voodoo Graphics, however, although it supported a subset of OpenGL to allow Quake to run, did not support a full OpenGL implementation until 1999, on the Voodoo 3. Thus, a major factor for

OpenGL's widespread use was the availability of a high-performance, high-quality, fully compliant OpenGL platform, in the form of the Riva TNT and its successors.

GeForce 256 (NVIDIA Corporation, 1999)

The next major breakthrough for consumer 3D hardware accelerators came in late 1999, with the introduction of the GeForce 256 accelerator by NVIDIA, which is still the state of the art. The most important property of the GeForce is that it features full geometry acceleration for the first time in the consumer price range. Previous accelerators were basically all rather simple rasterizers, in a way more 2D than 3D, where most of the 3D work had to be done by the driver, i.e. in software. With geometry acceleration, however, actual 3D primitives and transformation matrices get submitted to the hardware, which takes over most transformations, clipping, and projection, as well as rasterization of primitives.

By now, NVIDIA is probably the most important driving force behind the evolution of OpenGL and consequently the GeForce supports a huge number of OpenGL extensions. Among them, for instance, hardware bump mapping (which was first supported on a consumer card by the Matrox G400) and support of cubic environment maps. So, The GeForce hardware is apparently able to perform texture lookups into texture maps that actually consists of six textures (the six sides of the cube that represents the cubic environment map) in real-time.

The breakthrough achieved by the GeForce 256 is especially significant for computer graphics professionals, since it marks the time where consumer graphics hardware has definitely surpassed extremely expensive high-end workstation hardware of just a short time ago, with a product carrying a price tag that makes it even attractive for just playing computer games.

Figure 4.8 shows a highly-tessellated sphere environment-mapped using a cubic environment map, which is supported in hardware by the GeForce 256, and the corresponding environment map itself.

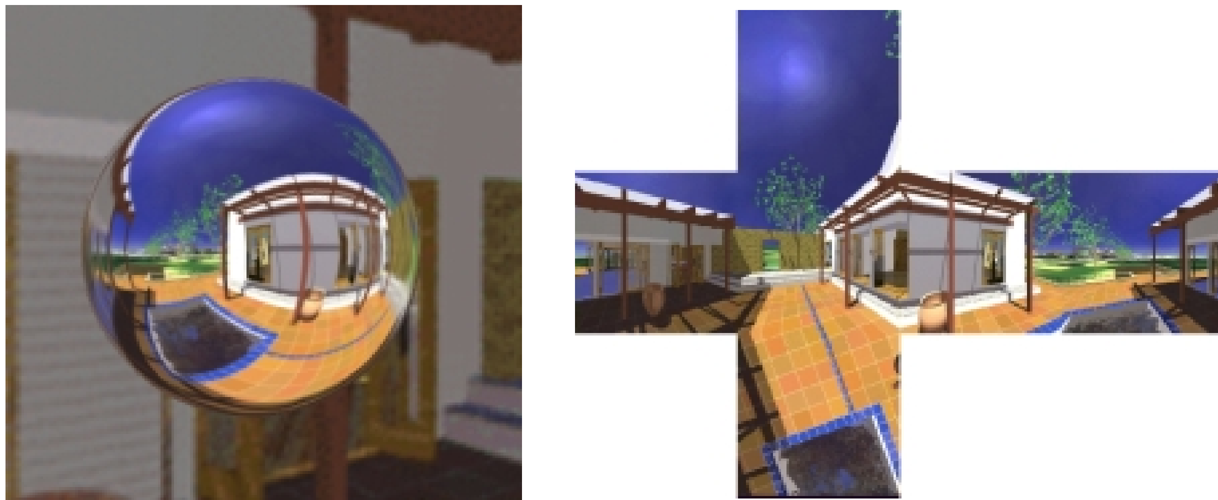

Figure 4.8: Cubic environment-mapping on NVIDIA's GeForce