Technical Report (12.06.2012)
Technische Universität München

RastaVox: Memory-Efficient Voxel-Model Rasterization

M. G. Chajdas[1] and M. Reitinger[1] and R. Westermann[1]

[1]Technische Universität München, Germany

Figure 1: *Views into the Crytek Sponza voxelized at a resolution of* $4096 \times 2048 \times 2048$. *The left view renders at 59 ms per frame (17 fps) on a GTX 680; the right view renders at 41 ms (24 fps) on a* 1280×720 *viewport with* $4\times$ *MSAA.*

Abstract

Voxel models are of increasing interest in 3D computer games, as they give rise to many efficient operations that can not be performed easily on adaptive surface models like triangle meshes. Examples include the efficient generation of level of detail hierarchies and model modifications like carving and extruding. However, to faithfully represent the surface and, thus, to render it in a similar quality than polygonal meshes, voxel representations require high spatial resolution along the surface. This typically implies that voxel models take up huge amounts of memory, making them unsuitable for computer games which have to run on fixed memory budgets.
In this report, we propose a novel approach for constructing, representing, and rendering voxel models efficiently on recent GPUs, at memory budgets on par with the requirements in computer games. We achieve this by adapting and extending classical iso-surface rendering techniques to modern GPUs, including the on-the-fly reconstruction of surface-aligned voxel models from volume data. Unlike previous techniques our approach utilizes the GPU's rasterization units for rendering. This makes it easy to integrate into existing rendering pipelines and allows taking advantage of hardware accelerated anti-aliasing. We demonstrate the construction of level of detail hierarchies for high quality rendering and real-time editing operations directly on our compact voxel representation.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Curve, surface, solid, and object representations

1. Introduction

Games and other 3D applications require increasingly detailed content. Most of this content is designed as textured polygonal meshes, or higher-order textured surfaces which

are triangulated for rendering. The increasing size of these meshes introduces a number of problems in the content creation pipeline; for instance, post-editing these meshes often requires expensive re-tesselation, and creating shape-preserving levels of detail is still a complex problem requiring additional, manual work.

Recently, voxel based representations have been introduced as a possible solution to overcome these problems [CNLE09, LK11]. Unlike polygonal meshes, voxel based representations resample both geometry and surface attributes like color into an uniform grid. This resolves many problems inherent to polygonal representations; for instance, models of a specific level of detail can now be generated efficiently on the voxel based representation, and texture resolution can be easily adjusted to match the geometric detail.

However, games must adhere to an additional set of constraints. For games, it is not sufficient to be able to render content quickly: Memory efficiency is an important concern, fast creation and modification has to be possible, and finally, the rendering technique should also easily integrate into the rendering pipeline.

Current techniques for voxel model rendering use the GPU's compute units to perform per-pixel ray-tracing. This makes it difficult to integrate into an existing rasterization based pipeline. For instance, anti-aliasing is a standard technique in rasterization to improve image quality; however, it is extremely expensive for ray-tracing—where it cannot be easily hardware accelerated. Using the rasterizer also allows easy integration with techniques like shadow maps, as can be see in Figure 1. Moreover, the ray-tracing methods only use the GPU's compute units, ignoring the fast blend and geometry processing hardware.

Furthermore, generation and especially dynamic updates of content are difficult to handle for voxel based rendering approaches. Either complex pre-processing is required [LK11], making it impossible for instance to generate the highest-resolution mesh and incrementally compute the level of detail from it, or the approach requires large amounts of memory as the geometry gets resampled into a sparse volume [CNLE09]. Even if the volume is already available, the current techniques require pre-processing to generate a level of detail hierarchy or an acceleration structure. Unfortunately, building an acceleration structure prevents further dynamic updates of the model.

Last, but quite important is the memory efficiency. In general, voxel based representations require a high-resolution grid in order to match the quality of a polygonal representation with textures. To render such grids efficiently, memory-intensive hierarchical acceleration structures are typically used.

In this paper we present a novel approach based on classic iso-surface rendering which is especially designed for the use in games: Our approach uses a memory-efficient representation, which only stores voxels along the mesh surface and renders those using the hardware rasterizer. We also show how this representation can be created quickly on the GPU from a volumetric representation or from a voxelization of a polygonal mesh, and how it can be modified in real-time, for instance, in order to apply decals. Finally, we describe how a level of detail representation can be computed directly on our compact representation on the GPU, requiring only small amounts of scratch space. Our technique allows for the rendering of large models at interactive frame rates, and is even fast enough to update, convert, and render a volume in real-time.

2. Related work

Voxel models have been first introduced 1993 by Kaufman in the seminal paper [KCY93]. Compared to polygonal representations, they provide an interesting set of advantages like easier level of detail computation and combined storage of surface and geometry information.

Recently, there has been a lot of research into large octrees to render such voxel models [CNLE09, LK11]. [CNLE09] subdivides the model into a sparse volume, storing only small volume "bricks" along the surface. It uses a compute based octree traversal to render the contained surface, and also supports fully volumetric rendering as required for instance for clouds. However, it has a significant memory overhead for solid models as it stores parts of the volume around the object surface. It also requires additional memory for the octree data structure on the GPU. [LK11] provides an interesting optimization by focusing on octress for surfaces: Along with the surface data like color, they also store contours which both improve the quality of the reconstructed surface as well as the performance of the rendering. In this case, the octree must be built top-down as successive levels combine the contours. Similar to GigaVoxels, the sparse voxel octrees also use the GPU's compute units for rendering.

Our representation builds upon the very first published work on iso-surface visualization: *Cuberilles* [HL79]. The Cuberille method—or opaque cubes—works by computing the set of grid cells that contain the iso-surface and rendering those as small cubes. The original method creates a single connected mesh during traversal to minimize the memory required by duplicated vertices. In order to improve the apparent surface quality, gouraud shading is used to interpolate the per-vertex normals along the surfaces.

A very popular method for rendering iso-surfaces is marching cubes. Marching cubes places vertices along the edges of each voxel and uses a variable amount of vertices for each of the 256 configurations [LC87]. This results in a high-quality surface approximation but comes with its own set of disadvantages. The biggest problem is memory usage. It is not uncommon that the marching cube mesh actually

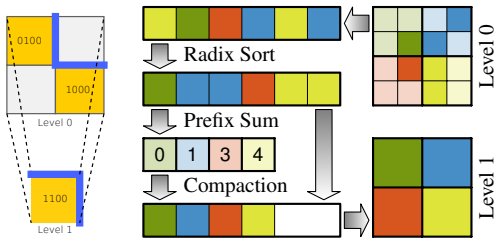

Figure 2: *Level of detail computation on our compressed representation. On the upper left, two surface voxels are highlighted in yellow, with their boundary faces in blue. The numbers indicate the bit-mask of which boundary faces are present. The faces of the merged surface voxel below are computed by using a bitwise* `or`*. On the right, the complete computation pipeline can be seen for the level of detail computation.*

exceeds the input volume in size, rendering them unsuitable as a compact representation of the volume. The efficient creation of marching cubes in parallel is also an issue, as the generated vertices are not independent but must be connected to form a mesh. For highly parallel architectures like GPUs, multiple passes over the data are required in order to obtain good performance [Gei07].

3. Algorithm

We introduce the notion of "surface voxels", which are voxels inside a volume that comprise the actual surface of the object. In this section, we will describe how surface voxels can be efficiently generated, how they can be used to compute a level of detail simplification and how they can be rendered efficiently using the hardware rasterizer.

Surface voxel generation

We assume that the input is provided as a voxel model, that is, as an uniform grid containing color, density and other attributes at each grid point. Starting from this grid, we first identify the boundary between "solid" and "empty" space. A boundary voxel is a voxel which itself is "solid" but has at least one "empty" neighbor in the 6-neighborhood. This results in one to six boundary faces. We store a bit-mask indicating which face is present and the position into a surface voxel buffer.

To make the storage more compact, we split input volumes into chunks of 256^3 or less voxels. This allows us to use only 8 bits per component for the voxel position. Together with the active face mask and padding, this results in 4 bytes per surface voxel. At this moment, the surface is already present, but the surface attributes like normals and colors are still missing.

The additional attributes are either stored per-face or per-voxel. In case of per-face data like normals, we create an additional buffer, store the per-face data into this buffer, and attach a pointer to the surface voxel. Constant per-voxel data is stored directly into the surface voxel buffer.

During the surface voxel generation, we have to constantly determine the next output slot as there is no one-to-one correspondence between input voxels and output surface voxels. We use global memory atomics for both the per-voxel and per-face buffers to determine the output slot. Even though surfaces are relatively sparse inside a volume—usually, less than 5% of the voxels—global memory atomics can become a bottleneck if the hardware does not provide a fast-path (see Table 2 for details.)

Level of detail

For level-of-detail, we use a multi-pass algorithm on the GPU which first sorts the data and then compacts each voxel in parallel. In the first step, we produce "runs" of surface voxels which all correspond to a single voxel at the next lower level. This can be done by performing a radix sort [Mer12] operation which ignores the last bit of each component of the position. Having the runs generated, we now have to find out where each run starts. This can be accomplished with a parallel prefix-scan [Mer12], which computes the start offset of each voxel run.

Finally, we compact each run into a single output voxel. We start one thread per run, which combines all its voxels: The boundary face bits of the children are `or`ed together. Per-voxel attributes are usually averaged at this stage; per-face attributes are combined for each face separately (see also Figure 2.) Combining the face bits using `or` is equal to performing a `min` operation on a density volume, that is, small features increase in size instead of disappearing as can be see in Figure 3.

The level of detail process requires an intermediate output buffer with a size equal to the input buffer, as the worst-case will contain one output voxel per input voxel. Compared to the input volume size, working on the surface voxel buffers results in an multiple order-of-magnitude reduction in memory usage.

Rendering

We use the geometry shader to generate cubes for each voxel. In order to be fast, we must ensure that the geometry shader produces as few triangles as possible. Otherwise, the amount of transient on-chip memory reduces the possible parallelism and decreases performance. The key observation is that if the surface is seen from the "empty" space, at most three faces of each voxel can be visible. This optimization assumes that the viewer can never enter the object or see "into" it—very similar to the requirements for meshes

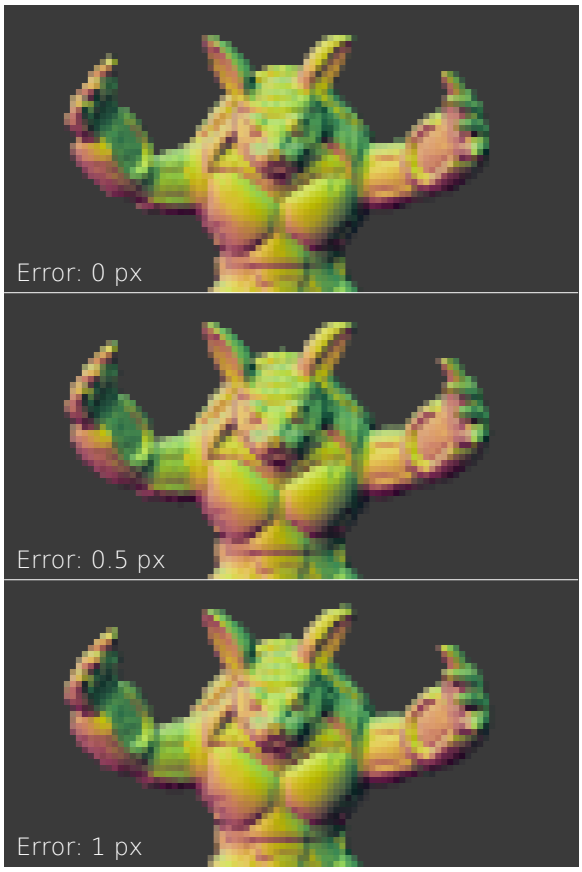

Figure 3: *Level of detail for a minified model. The* 2048^3 *Armadillo data set is seen from a great distance. At pixel error 0, the full resolution model is displayed, while at error 1, the highest level of detail approximation is selected. Due to the merging of surface voxels, the volume slightly expands with higher level of detail levels as can be seen for instance on the top of the head.*

that are rendered with back-face culling. If the viewer can enter the model, we have two possibilities: Interior faces can be marked as visible during the creation, resulting in a one voxel thick "shell" without reducing performance. Alternatively, the interior can be rendered by generating up to five faces per voxel at reduced performance. Throughout our testing, we always used the first method; the results can be seen in Figure 4.

Inside the vertex shader, we perform backface culling to determine which of the active faces are visible. This information, together with the position of the voxel, is then forwarded to the geometry shader. In the geometry shader, we generate up to three faces consisting of two triangles each. As we could have per-face data, we cannot share vertices between faces; the maximum number of vertices generated by the geometry shader is thus 12. The geometry shader also extracts and decompresses the normals, and optionally generates additional per-face attributes like UV coordinates.

Resolution	Base model	LoD	Total	ESVO
512	6.1	2.1	8.2	35
1024	24.9	8.4	33.3	95
1536	56.1	19	75.1	–
2048	99.9	33.9	133.8	243
4096	400	136.3	536.3	566

Table 1: *Memory size of the soldier model at varying resolutions in MiB. For comparison, we have included the matching octree sizes used by [LK11]. [LK11] provide higher-quality contours, but require similar subdivisions on this mesh to match the quality of the color data.*

4. Results

For rendering, we subdivide the input volume or voxel model into equally sized chunks. Unless noticed otherwise, we used 256^3 sized chunks. At runtime, we perform occlusion and view frustum culling to determine which chunks are visible. The occlusion culling is done using occlusion queries which render a solid cube for the whole chunk using the rasterizer and check for visibility in the next frame [Sek04]. If a chunk is determined to be visible, we estimate the worst-case projected area of a single voxel and select the level-of-detail depending on the allowed pixel error similar to [DSW09]. Notice that only the surface voxels are rendered and the provided resolutions are the resolution of the input voxelization including empty voxels.

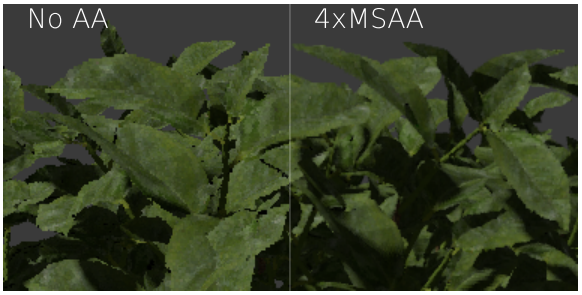

Figure 4: *A magnified view onto a complex bush mesh. The thin geometry and high geometric complexity is also present in the polygonal source model. Fortunately, our approach gets similar benefits from hardware MSAA as normal polygon meshes.*

Even large models can be quickly converted using our method. In Table 2, we can see that conversion performance mostly depends on the resulting surface complexity. In every case, faster atomic instructions—as present on the NVIDIA

GPU	Dataset	Base model	LoD	Total
GTX 480	Wood	25.2	37.8	63
	Hairball	205.3	216.5	421.8
	Armadillo	45.1	40.2	85.3
GTX 680	Wood	5.0	31.6	36.6
	Hairball	90.4	181.6	272
	Armadillo	31.3	39.9	71.2

Table 2: *Conversion performance for various models. The wood volume (*256^3*) contains density and color; while the armadillo (*2048^3*) and hairball (*1024^3*) volumes also store normals. All times are in ms.*

Dataset	Max error	No AA	4× MSAA
Buddha 4096^3	0	69.8	70
	1	18.7	19.3
	2	6.0	7.8
	4	5.6	5.5
Armadillo 2048^3	0	25.2	25.5
	1	20.0	19.9
	2	6.3	7.6
	4	3.3	3.3

Table 3: *Rendering performance for the Buddha* 4096^3*, Armadillo* 2048^3 *data sets on an NVIDIA GTX 680 at* 1280×720*. All times are in ms.*

GTX 680—help during the extraction phase. The level of detail step consists of multiple passes over the data and is mostly bandwidth limited.

In Figure 5, we can see the effect of increasing the maximum pixel error for a large voxel model, in this case, the Buddha statue at 4096^3 resolution. A pixel error of 1—while resulting in no visible difference—improves performance significantly from 70 ms down to 18 ms. At a pixel error of two, the first small inaccuracies along the silhouette become visible; however, the interior is still perfectly smooth. At a pixel error of four pixels, both the silhouette and the interior start to exhibit discretization artifacts while the performance improvement over a 2 pixel error becomes comparatively small.

Limitations

As our method uses discrete faces with per-face normals, artifacts become visible as soon as the projected voxel size becomes larger than a pixel on screen. Two main artifacts are visible: First of all, the silhouette exhibits block artifacts and second, the surface attributes are constant across a single face. The lack of interpolation can be partially resolved by using per-face textures or by using a screen-space blur, but the silhouettes cannot be cleaned up easily.

Resolution	No AA	4× MSAA	8× MSAA
512	5.0	5.0	5.1
1024	17.8	18.0	18.2
1536	38.6	38.8	39.2
2048	50.4	50.7	50.8
4096	42.0	43.1	43.4

Table 4: *Rendering performance for the Soldier model at different voxel model resolutions on an NVIDIA GTX 680 at* 1280×720*. At* 4096^3 *resolution, the level of detail efficiency improves as the chunks become smaller and have tighter bounds. All times are in ms.*

Those artifacts can be mitigated by computing the surface voxels at the correct level of detail. We expect that many games which have control over the minimum view distance—like strategy games—can easily replace parts of the world with voxelized representations while ensuring visual quality.

5. Conclusion and future work

In this paper we have proposed an efficient rendering technique for voxel based models. Unlike other current approaches, our method uses the hardware rasterization units and can thus be easily integrated into existing rendering pipelines. Our compact representation also allows the efficient generation of level of detail models. In a number of experiments we have demonstrated that our approach is suitable for rendering game content.

In the future, we would like to provide a higher-quality surface approximation which would allow close-up views. To achieve this, adaptive voxelization techniques which can dynamically generate a boundary voxel representation without a full voxel grid will be considered. In addition, the techniques described in [LK11] should be partially applicable to our method and allow for higher-quality silhouettes.

Acknowledgements

We would like to thank Jan Sommer for the help on the terrain and wooden box models and An Lu for help with the voxelization of the Sponza, Sibenik and "Soldier" scene. The soldier has been kindly provided by Alexandru Adrian Radoiu.

References

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (2009), I3D '09, ACM, pp. 15–22. doi:10.1145/1507149.1507152.

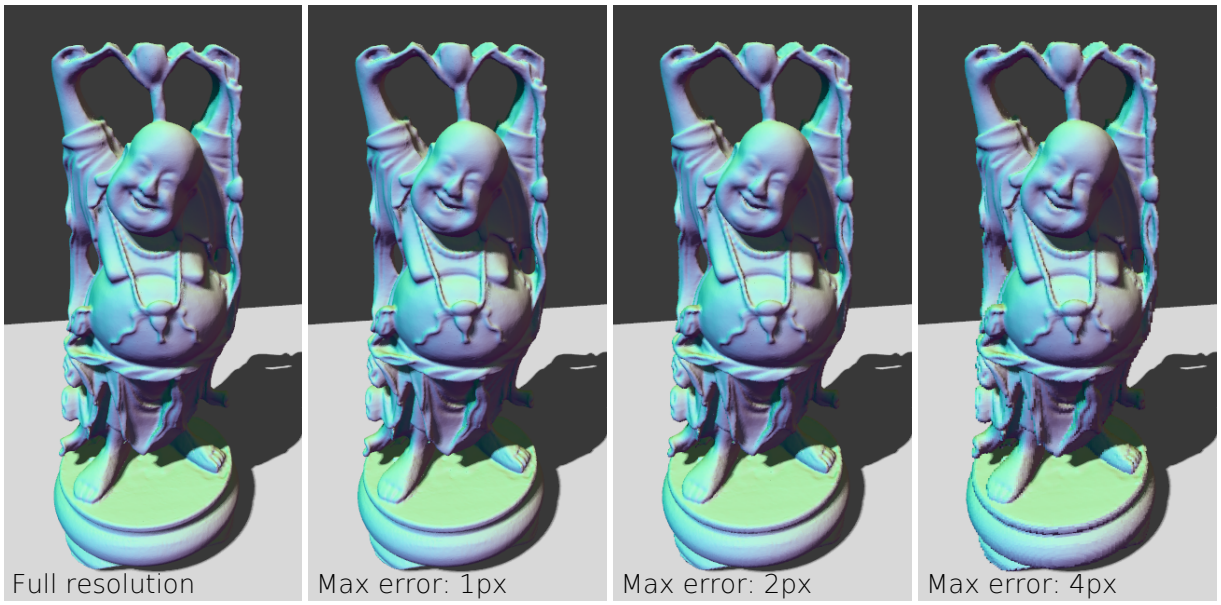

Figure 5: *Buddha* 4096^3 *data set rendered with different level of detail errors. The rendering performance varies between 70 ms per frame for the left-most image to 5.5 ms per frame for the rightmost at* $4\times$ *MSAA. Detailed timings can be found in Table 3.*

[DSW09] DICK C., SCHNEIDER J., WESTERMANN R.: Efficient geometry compression for gpu-based decoding in realtime terrain rendering. *Comput. Graph. Forum 28*, 1 (2009), 67–83.

[Gei07] GEISS R.: Generating complex procedural terrains using the GPU. In *GPU Gems 3*, Nguyen H., (Ed.). Addison-Wesley, 2007, pp. 7–37.

[HL79] HERMAN G. T., LIU H. K.: Three-dimensional display of human organs from computed tomograms. *Computer Graphics and Image Processing 9*, 1 (1979), 1–21. doi:10.1016/0146-664X(79)90079-0.

[KCY93] KAUFMAN A., COHEN D., YAGEL R.: Volume graphics. *Computer 26*, 7 (July 1993), 51–64. doi:10.1109/MC.1993.274942.

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3D surface construction algorithm. In *Computer Graphics (SIGGRAPH '87 Proceedings)* (July 1987), vol. 21, ACM, pp. 163–169. doi:10.1145/37401.37422.

[LK11] LAINE S., KARRAS T.: Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics 17*, 8 (2011), 1048–1059. doi:10.1109/TVCG.2010.240.

[Mer12] MERRY B.: CLOGS. MIT License, 2012. URL: http://sourceforge.net/apps/trac/clogs/.

[Sek04] SEKULIC D.: Efficient occlusion culling. In *GPU Gems*, Fernando R., (Ed.). 2004, pp. 487–503.

[SS10] SCHWARZ M., SEIDEL H.-P.: Fast parallel surface and solid voxelization on GPUs. *ACM Trans. Graph. 29*, 6 (Dec. 2010), 179:1–179:10. doi:10.1145/1882261.1866201.

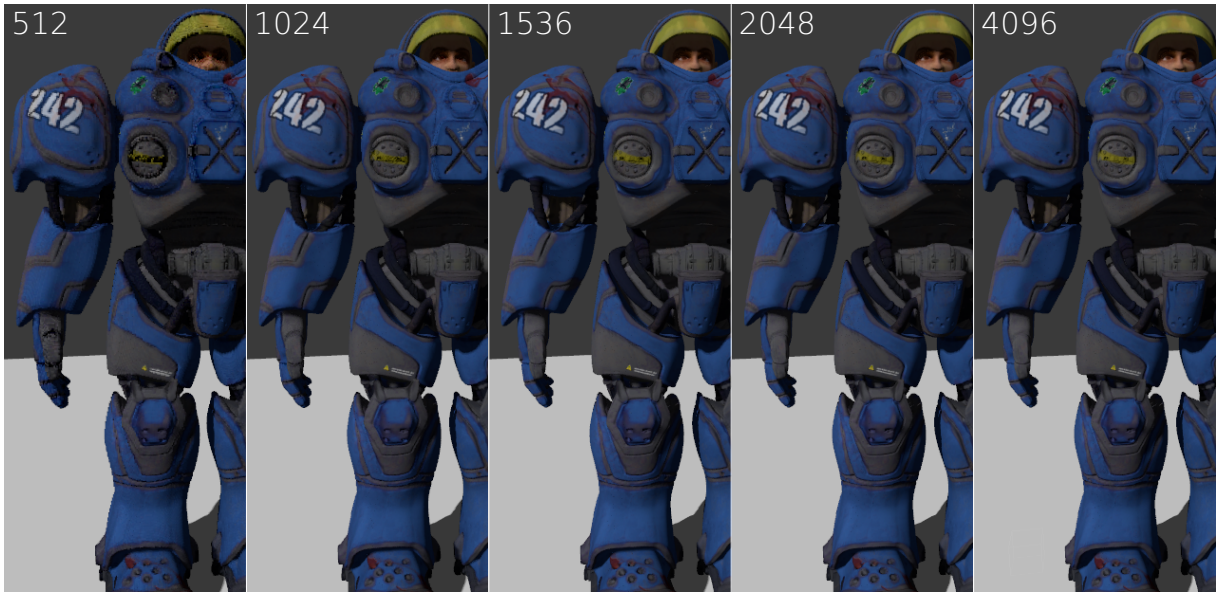

Figure 6: *The soldier model at various resolutions. The rendering performance varies from 5 ms for the leftmost image to 54.8 ms for the rightmost image on an NVIDIA GTX 680 at a* 1280×720 *viewport with* $4\times$ *MSAA. Detailed timings can be found in Table 4; model sizes can be found in Table 1.*