Legolizer: A Real-Time System for Modeling and Rendering LEGO® Representations of Boundary Models

Luís F.M.S. Silva
luisfmss@gmail.com

Vitor F. Pamplona
vitor@vitorpamplona.com
Instituto de Informática - UFRGS, Brazil

João L.D. Comba
comba@inf.ufrgs.br

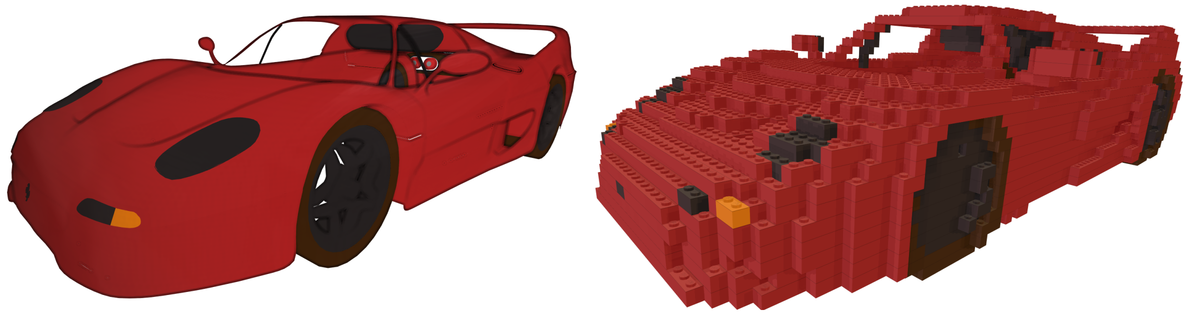


Figure 1. Ferrari example: a surface mesh (left) is converted and rendered into a LEGO® model (right) in real-time.

***Abstract*—In this work we propose a method for converting triangular meshes into LEGO® bricks through a voxel representation of boundary meshes. We present a novel voxelization approach that uses points sampled from a surface model to define which cubes (voxels) and their associated colors will compose the model. All steps of the algorithm were implemented on the GPU and real-time performance was achieved with satisfactory volumetric resolutions. Rendering results are illustrated using realistic graphics techniques such as screen space ambient occlusion and irradiance maps.**

***Keywords*-voxelization; GPU; LEGO® ;**

I. Introduction

LEGO® is a worldwide line of toys composed of colorful interlocking plastic bricks that can be assembled in many ways and positions. LEGO® toys were originally designed in the 1940s in Denmark and since then became very popular around the world. Throughout the years, the LEGO® trademark has achieved an extensive subculture that supports movies, contests, games (e.g. LEGO® Batman and LEGO® Star Wars), and four amusement parks [1]. In 2008, LEGO® was voted Britain's favourite toy [2], which proves its great popularity even after over 50 years of existence. Nowadays, some artists assemble exclusive objects using only LEGO® bricks [3], [4]. In this work we propose *Legolizer*, an algorithm that can help artists simulate such results by producing LEGO® -like renderings from triangular mesh models.

Our work focus on the cubic-like shapes predominant on the original LEGO® models, although many other shapes are often found in current toys. Therefore, the essence of our algorithm is a voxelization procedure, which receives as input a triangular mesh with colors associated to vertices, and produces as output a collection of bricks which are rendered in real-time using the GPU. There is a vast literature on voxelization algorithms for other purposes [5], [6], [7], [8], [9], [10], [11], [12]. The main difference of our approach is how we harness the processing power of GPUs to both generate and render the volumetric representations we generate. The original triangular mesh is first converted to a set of voxels, which are sent to a polygonization technique which generates uniform cubes. Every cube receives a little rounded cylinder on its top side, just like the interlocking part of the brick, while the bottom side is planar and closed. Bump mapping is used during rendering to simulate the relief of the LEGO® logo on top of each cylinder and to smooth the top corners, giving the appearance of joint pieces when a planar vertical wall is assembled. Figure 1 illustrates the results we obtain using Legolizer. The LEGO® representation is obtained through a volumetric model generated from the object surface.

The main contributions introduced in this work can be summarized as follows:

- Legolizer, the first algorithm to produce LEGO® -like representations directly from triangular meshes;
- A GPU-based voxelization and polygonization approach with color coherence to produce LEGO® bricks
- A rendering algorithm that process the volumetric representation of LEGO® bricks in real-time using screen

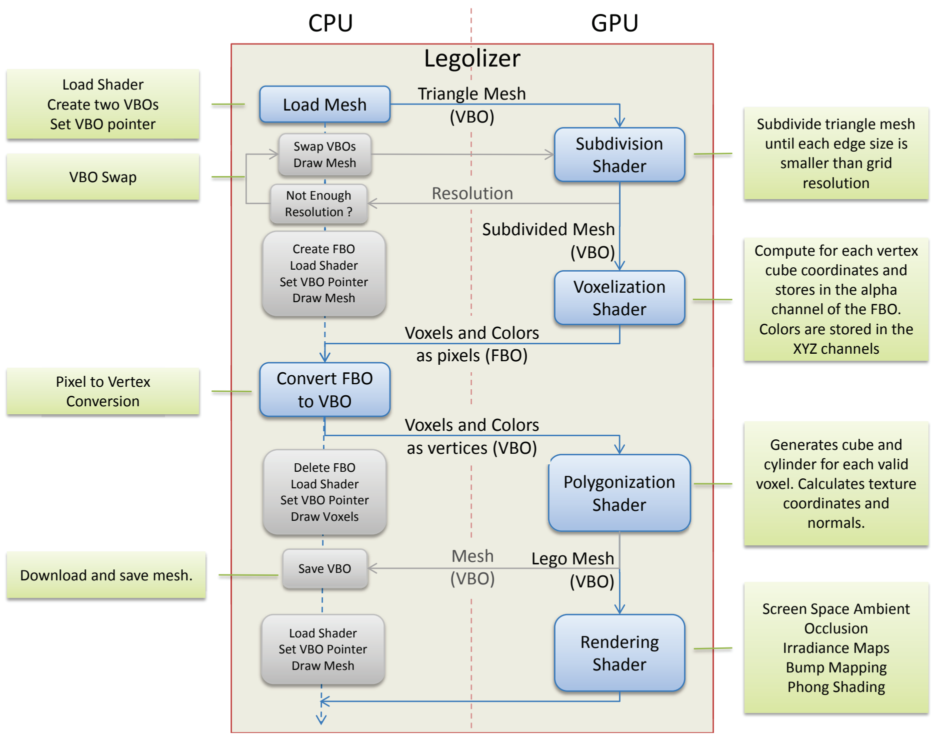


Figure 2. **Legolizer architecture** is composed of four main steps: (i) **subdivision**, where the triangular mesh is divided until a resolution criteria is met; (ii) **voxelization**, where each vertex is associated to one voxel; (iii) **polygonization**, where each voxel generates an uniform LEGO® cube; and (iv) **rendering**, where bump mapping is applied to give the final LEGO® appearance. Steps (i) (ii) and (iii) can be done in a pre-processing stage and the LEGO® mesh can be stored to speedup the rendering process.

space ambient occlusion and irradiance maps.

The paper is organized as follows. First we review the relevant work on voxelization algorithms in the literature. The Legolizer voxelization algorithm in all its steps is described in the section that follows. We show the results we obtained and conclude with directions for future work.

II. Related Work

Although voxelization methods have been studied since the 1980s, the first voxelization algorithms that use graphics hardware were developed only in the last decade, due to the flexibility in graphics hardware programming. For example, Fang and Chen [5], [6] proposed an algorithm that exploited the graphics hardware by slicing the object surface and using the sliced planes as clipping planes to reconstruct and render the object. Ogáyar *et al.* [7] adapted [8] to use the graphics hardware with standard OpenGL primitives and with the programmable capabilities of GPUs. Karabassi *et al.* [9] render six z-buffers, two per axis, for an object to reconstruct its surface in a grid. This approach is fast and simple to implement, but it misses concavities in convex objects. Passalis *et al.* [12] improved the algorithm proposed by Karabassi *et al.* [9] to support surface models of arbitrary topology without loosing performance.

Recently, Eisemann and Décoret [10] proposed a fast scene voxelization method using the graphics hardware, assuming that the rendered view of a scene implicitly defines a grid and keeping the depth value information encoded in the RGBA channels. This approach proved to be very fast and favorable to applications that used shadows and refraction. More recently, Eisemann and Décoret presented a single-pass technique to voxelize the interior of watertight 3D models [11] with high resolution grids in real-time.

Color information is not used in most voxelization applications, and therefore it is often discarded [5], [10], [11], [6], [9], [7], [12]. Our voxelization method has some similarities to the one developed by Karabassi *et al.* [9] and Passalis *et al.* [12], but while they use six z-buffers and thus six passes, our point-based approach uses only one render pass, and therefore uses a single buffer while keeping the original voxel color. Passalis *et al.* [12] also uses the graphics hardware to voxelization but they calculate the voxels on the CPU while we do it on the GPU.

III. Legolizer

The Legolizer algorithm is composed of a four-step process that creates and renders LEGO® representations from boundary models (Figure 2). The first step consists of a refinement in the triangular mesh using subdivision steps

Algorithm 1 Subdivision Technique (CPU)

```
Require: $M$ {Triangle Mesh}
 1: Create 2 VBOs: $vbo_1$, $vbo_2$
 2: Send $M$ to the GPU in $vbo_1$
 3: Set $vbo_1$ as the shader input and $vbo_2$ as output
 4: Enable subdivision shader
 5: while Vertex number in $M$ does not change do
 6:     Draw input VBO
 7:     Swap input and output VBOs
 8: end while
 9: Disable subdivision shader
10: return The last output VBO pointer
```

until the model is adequately sampled and therefore can create good voxelization results. This is governed by geometric criteria such as edge size, and are discussed in Section III-A. The second step of the algorithm consists in applying a voxelization procedure (Section III-B) over a triangular mesh to obtain a volume representation. The third step corresponds to a polygonization procedure (Section III-C) that generates LEGO® bricks with normal, texture and color information. The fourth step renders the LEGO® model by using modern real-time rendering techniques (Section III-D).

A. Step 1: Subdivision

The voxelization algorithm relies on a uniform sampling of the surface, which can not be assumed in general triangular meshes since triangles have irregular shapes and varying edge sizes. We used as assumption for our procedure that a triangle is well-sampled if its edges are smaller than a user-defined resolution. When this is not satisfied, we employ a subdivision strategy to partition the triangle into smaller triangles. The subdivision strategy used is very simple and discussed in several other works (see for instance [13]), and inserts vertices to partition edges that are larger than a given resolution. Since the voxelization technique we use is vertex-based, we must enforce that at least one vertex v of the input triangle mesh M is inside of each brick b of the LEGO®-based surface B for B to be watertight (*i.e.* without “holes“). In order to guarantee a watertight surface, it is necessary to subdivide the mesh M until each triangle edge e be smaller or equal than the cube edge length r.

The iterative nature of this algorithm is implemented as a multi-pass technique entirely on the GPU (Algorithm 1). Due to the inability to read-write textures on the GPU, we swap between two Vertex Buffer Objects (VBO), which serve as input and output buffers while the subdivision does not end. In the initial step, the input VBO is loaded with the mesh M. On each execution pass, vertices pass trough a simple vertex shader which emits its position and color. A geometry shader (Algorithm 2) splits the triangles into three different pre-defined patterns chosen by the number of edges greater than the cube size (Figure 3). The subdivision depends on the cube size and must be executed whenever this value changes.

Algorithm 2 Subdivision Geometry Shader (GPU)

```
Require: $v_1, v_2, v_3$ {Triangle Vertices}
Require: $r$ {Cube's size}
 1: $e \leftarrow$ Set of edges with length greater than $r$
 2: if $e = \emptyset$ then
 3:     Emit triangle $v_1, v_2, v_3$
 4: else if $|e| = 1$ {Figure 3(a)} then
 5:     $v_4 \leftarrow$ first vertex of $e_1$
 6:     $v_5 \leftarrow$ second vertex of $e_1$
 7:     $v_6 \leftarrow$ new vertex at the middle of edge $e_1$
 8:     $v_7 \leftarrow$ triangle vertex opposite to $e_1$
 9:     Emit triangle $v_4$,$v_6$,$v_7$
10:     Emit triangle $v_5$,$v_7$,$v_6$
11: else if $|e| = 2$ {Figure 3(b)} then
12:     $v_4 \leftarrow$ vertex shared by $e_1$ and $e_2$
13:     $v_5 \leftarrow$ vertex of $e_1$ different from $v4$
14:     $v_6 \leftarrow$ vertex of $e_2$ different from $v4$
15:     $v_7 \leftarrow$ new vertex in one half of $e_1$
16:     $v_8 \leftarrow$ new vertex in one half of $e_2$
17:     Emit triangle $v_4, v_7, v_8$
18:     Emit triangle $v_7, v_6, v_8$
19:     Emit triangle $v_5, v_6, v_7$
20: else if $|e| = 3$ {Figure 3(c)} then
21:     $v_4 \leftarrow$ new vertex in one half of $e_1$
22:     $v_5 \leftarrow$ new vertex in one half of $e_2$
23:     $v_6 \leftarrow$ new vertex in one half of $e_3$
24:     $v_7 \leftarrow$ vertex shared by $e_1$ and $e_2$
25:     $v_8 \leftarrow$ vertex shared by $e_2$ and $e_3$
26:     $v_9 \leftarrow$ vertex shared by $e_3$ and $e_1$
27:     Emit triangle $v_4, v_5, v_6$
28:     Emit triangle $v_7, v_4, v_5$
29:     Emit triangle $v_9, v_5, v_6$
30:     Emit triangle $v_8, v_6, v_4$
31: end if
```

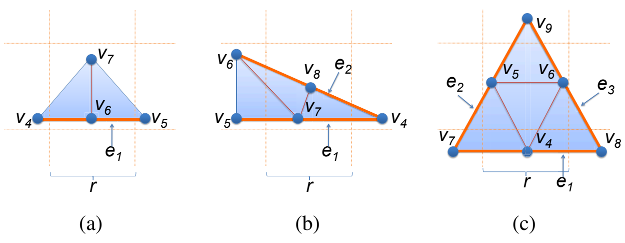


Figure 3. Subdivision cases when (a) one, (b) two, and (c) three edge lengths are greater than the cube edge size r. The shader divides the triangle according to the red lines.

B. Step 2: Voxelization

The voxelization uses the vertices of the refined mesh to find the locations to create voxels. For this we use an image-space approach. Given a viewing position, the algorithm renders all vertices of the original surface into a 2D image that is stored as a frame buffer object (FBO). This strategy relies on the fact that the projected vertices will sample well the projected surface, and therefore will allow for creating a good volumetric approximation.

The mesh is already on the GPU after the previous subdivision step, and passes through a shader that computes in which voxel each vertex will belong. Each vertex is projected into a FBO using an orthogonal camera (Algorithm 4). Voxels are placed inside a regular grid and sequentially

Algorithm 3 Voxelization (CPU)

```
Require: $M$ {Subdivided Triangle Mesh}
1: Create FBO
2: Set ortogonal projection to FBO
3: Enable mapping shader
4: Draw $M$, which already is on GPU
5: Disable mapping shader
6: Download FBO
7: Create a VBO from FBO
8: return VBO pointer
```

Algorithm 4 Voxelization: Vertex Shader

```
Require: $vp$ {Vertex 3D Position}
Require: $vc$ {Vertex Color (RGB)}
Require: $size$ {Grid size}
1: $voxel_{xyz} \leftarrow vc_{xyz}$
2: $voxel_w \leftarrow int(vp_x) \times size \times size$
          $+ int(vp_y) \times size$
          $+ int(vp_z)$
3: if $voxel_w$ is outside the grid then
4:     $voxel_w \leftarrow$ NULL
5: end if
6: $voxel_w \leftarrow voxel_w/(size * size * size)$
7: Emits the $voxel$ as a pixel
```

numbered according to their 3D position, in axis ordering: x, y, and z. When a vertex is placed outside the grid, it receives a null value as voxel number. The FBO contains information about the vertex color in the x, y, z channels and the normalized voxel number in the alpha channel. The FBO is downloaded to the CPU and re-sent as a VBO, where each pixel becomes a vertex.

Obviously, some vertices are written at the same position in the FBO or discarded by the Z-buffer test. This entails two important characteristics of our proposal: (i) unlike the method developed by Karabassi *et al.* [9] which renders the surface six times, we use a single frame buffer and only one rasterization is needed; and (ii) the front surface is completely displayed, but back surfaces may contain holes. Fortunately, these holes can be avoided by increasing the frame buffer size. In Section IV we provide an analisys of the method performance as function of the FBO size.

C. Step 3: Polygonization

Given the voxel information stored in the VBO, numbers (stored in the alpha coordinate) and colors (stored at the x, y, z coordinates), we use the geometry shader to generate triangles of the LEGO® bricks (Algorithm 5). The alpha channel (w) is first converted to spatial coordinates representing the center of the cube. Given the cube edge length r, the shader computes each face of the cube with normal and texture coordinates, discarding the vertices with ilegal values in the alpha channel. Remember that, in the step of mapping vertex to voxels, we set NULL values in the alpha channel for vertices outside the grid, thus making the models partially or entirely voxelized depending on its position inside the grid.

Algorithm 5 Polygonization: Geometry

```
Require: $v$ {Vertex}
Require: $size$ {Grid size}
Require: $r$ {Cube size}
1: if $v_w$ not NULL then
2:     $v_w \leftarrow v_w \times (size \times size \times size)$
3:     Load $x, y, z$ cube center coordinates from $v_w$
4:     Set output color as $v_{xyz}$
5:     Given $x, y, z$ and $r$, Emit as a triangle strip:
6:         6 cube faces with normal and tex. coord.
7:         1 cylinder above the cube top face
8:         1 disk with normal and tex. coord. to cover the cylinder
9: end if
```

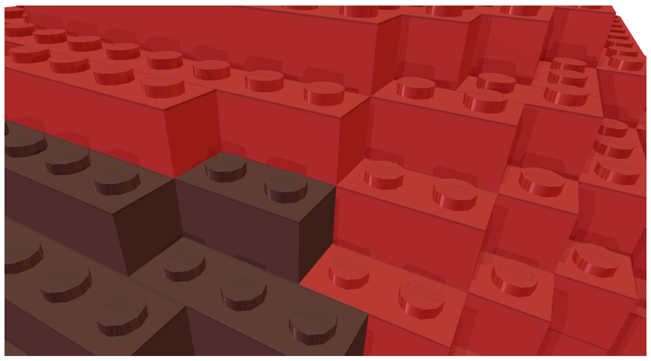


Figure 4. LEGO® bricks rendering

This method does not guarantee that only one cube will be generated for each voxel. Instead, we let the graphics pipeline remove the duplicated cubes in the same spatial position by culling, clipping and z-tests. We did not experienced z-fighting between duplicated cubes in our tests.

D. Step 4: Rendering

The rendering of LEGO® bricks (Section III-D) uses Phong shading [14] and a real-time screen space ambient occlusion [15]. Ambient occlusion is an approximation to global illumination and is defined as the amount of occlusion neighboring occluders define over a point. Together with the occlusion, the mean direction of incident light called band normal is computed. Since, ambient occlusion is a computationally expensive technique, it is necessary to perform some approximations to allow real-time rendering. One is the screen space ambient occlusion [16], which we implemented in our LEGO® representation. This approach is independent from scene complexity and approximates the occlusion caused by near and distant surfaces.

Bump mapping was applied on the disk covering the cylinders to simulate the relief of the LEGO® symbol, and over each vertical cube face to smooth the top corner normals, generating the appearance of interlocking pieces (Figure 4).

IV. Results

In this section we summarize the results we obtained. Tests were performed on a AMD Athlon 64 3500+ with 2GB

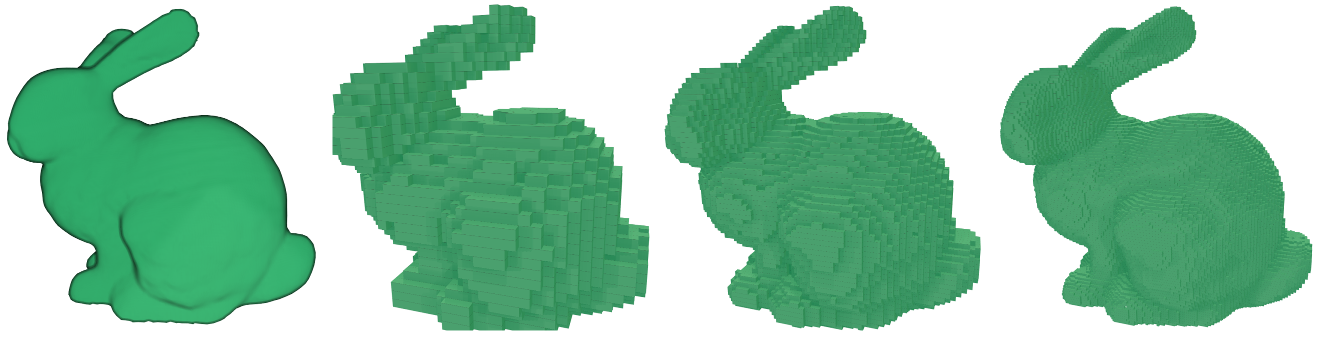


Figure 5. Bunny voxelization generated with 1024x1024 of frame buffer size. a) original mesh b) 32^3 resolution; c) 64^3 resolution; d) 128^3 resolution;

of RAM, and a GeForce 8800GTX on a Linux 32 bits. As mentioned before, our voxelization approach mainly depends on the frame buffer size. It represents a tradeoff between performance and accuracy. However, the model shape also affects the accuracy, since it can be prone to generate overlapped points, thus demanding a bigger frame buffer to achieve good results. Usually, a frame buffer with size of 512x512 pixels produces good results with resolutions up to 128^3 voxels. In complex objects it is necessary to increase the frame buffer to values over 1024x1024 pixels. Figure 5 shows the voxelization of model Bunny in different resolutions using a frame buffer size of 1024x1024.

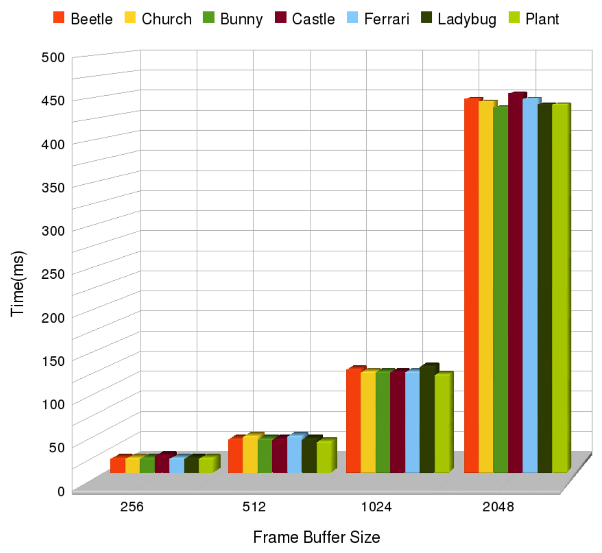
Figure 6.(a) shows how the time to generate each frame increases when the frame buffer size grows. This happens due to two reasons: a bigger frame buffer results in more data to be sent to the GPU and more vertices to be processed, thus increasing the number of threads. Our method tends to be up to 5 times faster than Passalis *et al.* [12], since we render the model only once and use a single frame buffer, while they renders the model six times and uses more than six buffers. However, our approach depends on the model surface complexity, thus producing different resulting times, opposed to Passalis *et al.* [12] that is almost surface independent and presents similar results with different models. Karabassi *et al.* [9] missed concavities in concave objects and Passalis *et al.* [12] missed objects that are not strictly closed or with irregular faces. However, as points from inside surfaces are also projected, our voxelization method does not suffer from these issues and voxelizes models with or without holes, including its internal parts.

The model is rendered in an environment illuminated by high dynamic range (HDR) light probes which generates irradiance maps to be used as the ambient color of the Phong equation sampled by the band normal of the ambient occlusion technique. The final LEGO® bricks rendering is showed in Figure 4 and the final Legolizer output is presented in Figure 7. After applying all these techniques we rendered the models with different resolutions, and achieved real-time performance. Figure 6.(b) shows the FPS for different models in different resolutions. Notice that even for 128^3 resolution, all datasets are rendered in real-time.

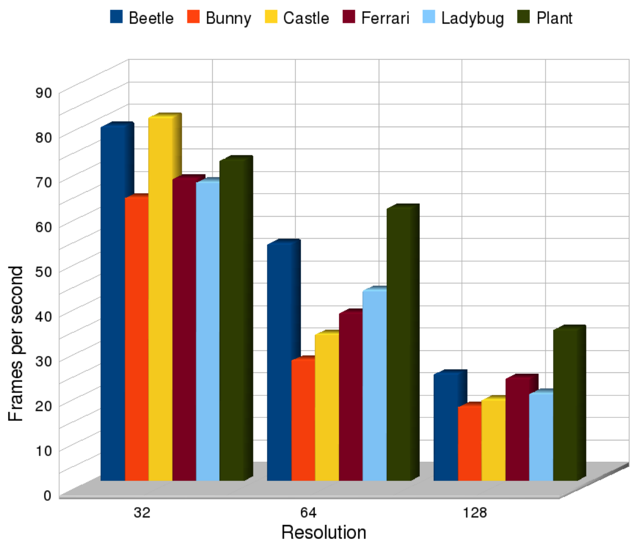
V. Conclusions and Future Work

In this work we presented a real-time method to convert and render LEGO® representations from surface models. Along this work we developed a new voxelization approach, which proved to be fast and flexible. It can be seen as an alternative to Passalis *et al.* [12], but faster and with precision dependent of the frame buffer size. This approach fits perfectly on our LEGO® conversion as it has good performance, provides voxel colors, and can adjust the accuracy according to our needs.

Although our method achieved real-time performance, there are some improvements that could be developed to save memory and processing time. The first modification concerns the way data is saved on a texture to store voxelization and color information. We used a four channels(RGBA) 32-bits precision texture, three channels(RGB) for colors and one(A) for the voxelization. However, a single channel could be used by compacting the data into a single 32-bits channel. Furthermore, regarding primitive generation, there could be developed a method to sample a unique vertex inside each voxel to generate a cube. This would avoid the need to generate and remove duplicated cubes. Another improvement can be made in the cylinders generation, since they represent the major challenge to render our models with real-time performance due to the great amount of vertices inserted. Cylinders are generated along with cubes, thus for each cube there is a cylinder. Their generation could be split and an algorithm to determine voxels with upper neighbors could be applied to detect hidden cylinders, thus saving both memory and processing time. This work can also be used together with a Microsoft proprietary technique on automated brick layout[17] which provides a step by step method to assembly the object from a brick(voxel) representation.



(a) Voxelization time



(b) Rendering FPS

Figure 6. Performance results: (a) time (in milliseconds) elapsed with the frame buffer size increase for several different datasets and (b) the frame rate obtained for different grid resolutions. Notice that results in (a) show that the frame buffer size affects all the models in the same manner. In (b) we can notice how the grid resolution affects the frame rate performance for each dataset.

Acknowledgment

The authors acknowledge Manuel M. de Oliveira Neto for interesting discussions and ideas for this work. Luís F.M.S. Silva, Vitor F. Pamplona and João L. D. Comba are sponsored by CNPq-Brazil grants (131322/2008-7, 142563/2008-0, 485853/2007-8 and 569239/2008-7).

References

[1] Vaunt Design Group, “Lego history: Invention of lego,” 2005. [Online]. Available: http://tinyurl.com/pljydl

[2] Telegraph Media Group Limited, “Lego named britain’s favourite toy,” 2008. [Online]. Available: http://tinyurl.com/p4scvw

[3] N. Sawaya, “Nathan sawaya: The art of the brick,” 2009. [Online]. Available: http://www.brickartist.com/

[4] S. Kenney, “Sean kenney: Art with lego bricks,” 2009. [Online]. Available: http://www.seankenney.com/

[5] H. Chen and S. Fang, “Fast voxelization of three-dimensional synthetic objects,” *J. Graph. Tools*, vol. 3(4), pp. 33–45, 1998.

[6] S. Fang, S. Fang, H. Chen, and H. Chen, “Hardware accelerated voxelization,” *Comput. Graph.*, vol. 24, pp. 200–0, 2000.

[7] C. J. Ogáyar, A. J. Rueda, R. J. Segura, and F. R. Feito, “Fast and simple hardware accelerated voxelizations using simplicial coverings,” *Vis. Comput.*, vol. 23, no. 8, pp. 535–543, 2007.

[8] A. J. Rueda, R. J. Segura, F. R. Feito, and J. R. de Miras, “Rasterizing complex polygons without tessellations,” *Graph. Models*, vol. 66, no. 3, pp. 127–132, 2004.

[9] E.-A. Karabassi, G. Papaioannou, and T. Theoharis, “A fast depth-buffer-based voxelization algorithm,” *J. Graph. Tools*, vol. 4, no. 4, pp. 5–10, 1999.

[10] E. Eisemann and X. Décoret, “Fast scene voxelization and applications,” in *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM, 2006, pp. 71–78. [Online]. Available: http://artis.imag.fr/Publications/2006/ED06

[11] E. Eisemann and X. Décoret, “Single-pass gpu solid voxelization for real-time applications,” in *GI ’08: Proc. of Graphics Interface 2008*. Toronto, Ont., Canada, Canada: Canadian Information Processing Society, 2008, pp. 73–80.

[12] G. Passalis, I. A. Kakadiaris, and T. Theoharis, “Efficient hardware voxelization,” in *CGI ’04: Proc. of the Computer Graphics International*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 374–377.

[13] W. F. Engel, *ShaderX 7: Advanced Rendering Techniques*. Charles River Media, 2009.

[14] T. Whitted, “An improved illumination model for shaded display,” *Commun. ACM*, vol. 23, no. 6, pp. 343–349, 1980.

[15] *Finding next gen: CryEngine 2*. New York, NY, USA: ACM, 2007.

[16] P. Shanmugam and O. Arikan, “Hardware accelerated ambient occlusion techniques on gpus,” in *I3D ’07: Proc. of the 2007 Symposium on Interactive 3D graphics and games*. New York, NY, USA: ACM, 2007, pp. 73–80.

[17] D. V. Winkler, “Automated brick layout,” 2005. [Online]. Available: http://tinyurl.com/plutj8

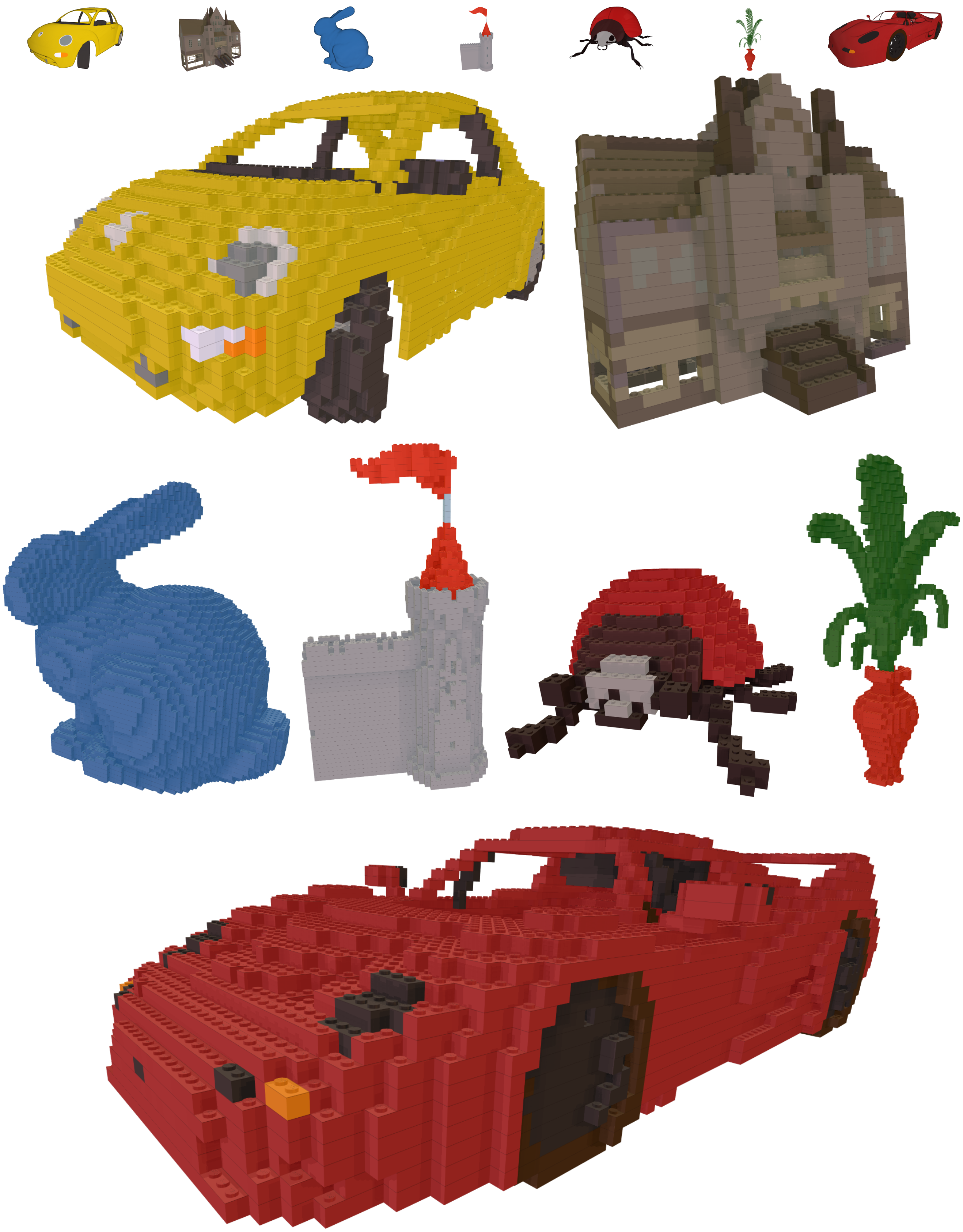


Figure 7. LEGO® representations created by our method