Unlimited Object Instancing in real-time

Szymon Jabłoński
Institute of Computer Science
Warsaw University of Technology
ul. Nowowiejska 15/19
00-665 Warsaw, Poland
s.jablonski@ii.pw.edu.pl

Tomasz Martyn
Institute of Computer Science
Warsaw University of Technology
ul. Nowowiejska 15/19
00-665 Warsaw, Poland
martyn@ii.pw.edu.pl

ABSTRACT

In this paper, we propose a novel approach to efficient rendering of an unlimited number of 3D objects in real-time. We present a rendering pipeline that is based on a new computer graphics programming paradigm implementing a holistic approach to the virtual scene definition. Using Signed Distance Functions (SDF) for a virtual scene representation, we managed to control the content and complexity of the virtual scene with the use of mathematical equations. In order to solve the limited hardware problem, especially the limited capacity of the GPU memory, we propose a scene element repository which extends the idea of the data based amplification. The content of the repository strongly depends on a 3D object visualization method. One of the most important requirements of the developed pipeline is the possibility to render 3D objects created by artists. In order to achieve that, the object visualization method uses Sparse Voxel Octree (SVO) ray casting. The developed rendering pipeline is fully compatible with the available SVO algorithms. We show how to avoid occlusion errors which can occur in the SDF and SVO integration single-pass rendering pipeline. Finally, in order to control the content and complexity of the virtual scenes in an unlimited way, we propose a collection of global operators applicable to the virtual scene distance function. Developed Unlimited Object Instancing rendering pipeline can be easily integrated with traditional visualization methods, e.g. the triangle rasterization. The only hardware requirement for our approach is the support for compute shaders or any GPGPU API.

Keywords

Computer graphics, voxel rendering, sparse voxel octree, signed distance function, instancing, data based amplification, holistic programming paradigm, procedural graphics, level of detail

1 INTRODUCTION

Signed Distance Functions (SDF) derive from fractal theory and their application to computer graphics originated with a method of ray-tracing quaternion Julia sets [Hart89]. Among others, the paper showed that SDF relatively simple equations can represent highly detailed, complex geometry. This opened the door to modeling and rendering very complex virtual scenes. Unfortunately, the limitations of the then hardware did not allow for the full use of the new approach. Except for the Julia sets, SDF-based modeling and visualization methods were usually limited to virtual scenes consisting of only basic primitives and relatively simple isosurfaces. In this paper, we present an algorithm that significantly extends the idea of SDF based visualization in combination with the Sparse Voxel Octree 3D object representation.

One of the most common indicators used to evaluate the quality of the computer graphics is the virtual scene complexity. In order to achieve realistic rendering results, the virtual scene must be represented as a collection of high-resolution 3D objects with detailed geometries and materials.

In order to render complex scenes containing numerous high-resolution 3D objects in real-time, the rendering pipeline should be based on the three important types of algorithms. The first one refers to the virtual scene management. In order to process culling operations (e.g. the frustum inclusion test) in an efficient way, we need to organize the scene using some sort of a hierarchical spatial structure [Cao10]. The second type of an algorithm deals with the level of detail (LOD) management [Lueb02]. Using defined LOD evaluation functions, we can select which visible objects should be rendered with higher LOD and which could be rendered without some details. Finally, the third type of an algorithm is an instancing algorithm that is used to render many instances of objects stored in the GPU memory usually along with an affine transformation and material parameters associated with each instance [Suther63].

Thanks to the constantly increasing computation power and memory capacity of today's GPUs, we can process and render more and more polygons per frame. At the same time, however, the complexity of objects in virtual scenes is also increased. It means we must improve the algorithms, which are used in computer graphics engines, so as to increase the complexity of virtual scenes.

In this paper, we present a novel approach to an efficient real-time rendering of a potentially unlimited number of 3D objects. By using Signed Distance Functions for the virtual scene representation, we extend the idea of the object instancing. Thanks to the SDF representation integrated with Sparse Voxel Octrees (SVO), we are able to render as many 3D objects created by artists as we want in real-time. The foundation of the developed algorithm is a novel computer graphics paradigm which we called *Holistic Graphics Programming*. By redefining the notion of the virtual scene, we developed a method for the virtual scene LOD management for an unlimited number of 3D object—Unlimited Object Instancing rendering pipeline (UOI).

2 RELATED WORK

There is a wide selection of literature about each mentioned component of the developed UOI rendering pipeline.

Over the years, many methods of the scene representation, LOD management and object instancing have been developed. Also, there are many papers on the Signed Distance Functions and Sparse Voxel Octrees. However, there is no related work in the context of UOI rendering pipeline algorithms that can be created by integrating this idea with the holistic approach to the scene representation, particularly in the context of visualizing 3D objects created by artists. Therefore, below we focus mainly on papers that are directly related to each component of our approach.

As mentioned in the introduction, the most common method to increase the complexity of the virtual scene is the object instancing [Suther63]. The object instancing is quite an old approach that can be placed in the context of various visualization methods. The main idea of the instancing is a compression of the virtual scene description. Instead of storing an information about the geometries of all instances of a given object in the scene separately (e.g., of each tree in a forest scene), it is possible to store only the geometry data of a single object and an additional buffer for the data that individualizes the instances once they placed in the scene (e.g., transformations that define the localization, size and orientation of an instance in the world coordinates).

Deussen et al. presented a great example of how to exploit the instancing approach to create realistic plant ecosystems in non-real-time graphics engines [Deussen98]. The geometry instancing approach is very popular for creating realistic botanical scenes due to the nature of plant structure with numerous similar elements [Snyder87, Hart91, Hart92, Kay86], and is commonly utilized in computer games. Using the instancing approach it is possible to render many instances of a given source object. However, in order to create a realistic, highly detailed virtual scene, each instance should have unique attributes. For this goal the data based amplification approach can be used to procedurally generate a variation of instances. Procedural noises and random functions can also be used to create unique, detailed variations of instances objects on the fly.

The modern GPU APIs offer a hardware-accelerated functionality for instancing geometry in real-time. Martyn showed how it can be utilized in the context of the self-affine geometry of IFS for real-time visualization of fractals [Martyn10]. Nevertheless, virtual scene rendering with the geometry instancing is limited to low-poly objects in real-time. It can still be used to render botanical scenes in an efficient way, but there is a problem to process many instances of high-resolution objects in real-time even by means of today's GPUs. The reason is that the geometry instancing with the triangle rasterization pipeline is limited by the object-space computation complexity.

Signed Distance Functions are widely used in the computer graphics from modeling and visualization of fractals [Reiner11], soft shadow generation [Wright15, Keinert14] to the font rendering [Green07]. One of the first paper that utilized this method of an object representation in the context of visualization was [Hart89]. It presented the idea of unbouding volumes which were used to ray-trace quaternion Julia sets. The idea was later extended by Hart et al. into the so-called Sphere Tracing [Hart94, Hart97].

Given an object represented by the distance function, sphere tracing relies on an iterative traversing a ray from the eye through the projection plane towards the object. For each iteration, we calculate an SDF estimated distance to the object, and if the estimation is smaller than a predefined value, the ray is considered to hit the object.

For a single primitive object like AABB, the classic ray–AABB intersection test will be much faster, because there is no need to perform many distance estimations presented in sphere tracing. However, SDF functions can be used to create highly detailed procedural objects using SDF primitives with boolean operators. Reiner et al. presented an introduction to an interactive SDF ray marching pipeline with a procedural object generation based on domain operations [Reiner11].

In the context of our work, the a most interesting operation that can be applied to distance functions is the object repetition generated by the modulo function in either controlled or unlimited manner.

Thanks to the increased computation power of today's GPUs and newly developed Sparse Voxel Octree algorithms, the high-resolution voxel-based representation is now ready for real-time applications. Due to the screen-space computation complexity of the SVO rendering pipeline, numerous high-resolution 3D objects can be processed in real-time using instancing approach. Cyril Crassin was able to perform visualization of the global illumination using SVO and voxel cone tracing [Crassin11]. There are also a few promising implementations of efficient ray tracing of SVO [Laine10] and even object animation, deformation and fracturing in real-time [Bau11, Wil13, Domaradzki16]. For that reason, the utilization of the SVO-based representation seems to be a very promising solution for modeling and visualization of 3D high-resolution objects. Moreover, SVO offers a continuous and symmetrical method of the LOD transition without any visible transition artifacts [Jab16].

3 UNLIMITED OBJECT INSTANCING

In this section, we describe requirements of the UOI rendering pipeline in real-time. Keeping in mind the hardware limitations like the limited capacity of the memory or the computation precision, we need to start with the proper definition of the UOI rendering pipeline.

3.1 Unlimited Object Instancing definition

For the purposes of this work, the requirements and features of the UOI rendering pipeline are defined as follows:

- A real-time rendering pipeline that is able to process and render an unlimited number of objects in the virtual scene. If it is possible to store a 3D object in the GPU memory and render it in real-time, it should be possible to store and render potentially unlimited instances of this object with unique variations without any noticeable performance hit or memory requirements increase.
- The possibility of visualizing 3D objects which were created by artists.
- A continuous and symmetrical LOD management of the virtual scene.

Considering the requirements above, it may seem that the most serious development obstacle is the hardware limitation. However, in our opinion, this is not the main problem. Computer hardware is and, presumably, will always be limited. In order to develop the UOI rendering pipeline, it's necessary to change the current computer graphics programming paradigm, for example, by applying the holistic approach to the definition of the virtual scene.

However, before we present the idea of the UOI rendering pipeline, we analyze some possible, naive designs that could be created using the available algorithms. Doing so, one can exclude algorithms and structures that cannot be used in the context of the proper implementation of the Unlimited Object Instancing pipeline.

3.2 Naive Unlimited Object Instancing

For the polygonal representation of geometry, the hardware instancing functionality, which is implemented in all modern GPUs, can be used. On the other hand, for a rendering pipeline that offers screen-space computation complexity like e.g. ray-casting, the software instancing approach can be used to render numerous instances of 3D objects. Both methods can be used to render many instances of a given collection of 3D objects. However, memory requirements for the virtual scene description would be increasing significantly because of the need of storing a unique data for each instance (e.g. model-to-world space transformations).

The second possibility is to store the whole scene in a single spatial structure like e.g. Sparse Voxel Octree. Using the DAG algorithm it is possible to store and render a scene object represented by a high-resolution grid (even of 128^3 resolution) [Kampe13] in real-time. The high-resolution scene grid could be also used as a virtual scene description. Then, 3D objects instancing could be used. However, none of them could be directly used to develop the proper UOI Instancing rendering pipeline.

3.3 Issues with classic paradigm

In all modern games and virtual simulations taking place in a 3D world, the virtual scene is defined as a collection of objects. In order to process and render a complex scene based on such a paradigm, it is necessary to check out which 3D objects are visible from the current point of view. Moreover, to execute the frustum visibility and occlusion tests in an efficient way, it is necessary to organize the objects implementing some sort of a hierarchical spatial structure.

This standard approach to the scene representation defines the classic real-time computer graphics programming paradigm. For the purpose of our work, we called it the *Object-Based Graphics Programming*. The main features of this programming paradigm are:

- A virtual scene is defined as a collection of 3D objects.

- The visibility and occlusion tests are executed to classify objects to render.
- Hierarchical spatial structures are used to organize the virtual scene and to increase the performance of visibility tests.

Due to hardware limitations, it is neither possible to store an unlimited collection of objects in the memory nor to perform the visibility tests for all of them. In order to develop the UOI rendering pipeline, we need to change the computer graphics programming paradigm to the paradigm which we called the *Holistic Graphics Programming*.

4 HOLISTIC UNLIMITED OBJECT INSTANCING

The foundation of the UOI rendering pipeline is the holistic approach applied to the virtual scene definition. In the holistic architecture, we replace a collection of objects with a single object—just the virtual scene object. By controlling the LOD of the virtual scene object, we will control the content and complexity of the scene regarded as a single entity, processing potentially an unlimited number of objects.

The example of the holistic approach can be found in a few popular computer graphics algorithms. A great example is the procedural terrain rendering, especially when considered in the context of its local LOD management. Using tessellation shaders, the geometrical complexity of the terrain geometry can be locally increased or decreased by controlling a tessellation factor for each patch of the terrain independently. The similar approach is used in the holistic UOI rendering pipeline to control the LOD of a scene.

The UOI rendering pipeline we developed is based on the following four components:

1. **Graphic asset repository**
2. **Virtual Scene representation**
3. **3D object visualization**
4. **Global operators collection**

4.1 Graphic asset repository

The *Graphic asset repository* contains a collection of all visual ingredients which the scene is composed of. In order to render the scene in real-time without using the data streaming functionality, all visual assets like 3D geometries and materials need to be stored in the GPU memory. The type and format of this data depend on the visualization method. In our implementation, by default, it is a collection of SVOs and textures.

Due to the limited memory capacity of GPUs, the repository contains a finite number of elements. Then, using the instancing approach with a data amplification method, the elements are rendered with unique variations, creating a complex, detailed virtual scene.

4.2 Virtual scene representation

The second component of the UOI rendering pipeline is the way we apply the holistic approach to the scene representation. In the proposed solution, we have implemented it by means of Signed Distance Function—the whole scene is represented by a signed distance function.

Since we use SDF, it is possible to create a highly detailed complex scene from primitive distance functions along with boolean operators. For the purpose of this work, we do not use complex distance functions, and the basic scene element is defined as the distance function for the cube primitive:

$$d = length(max(abs(p+o)-b),0) \qquad (1)$$

where:

d	= distance to the object
p	= point on ray from the eye
o	= offset from scene origin
b	= cube size

Using the SDF ray marching, we can generate cube primitives efficiently. After that, each cube primitive could be replaced with the 3D object represented by the SVO. On the basis of the resulting nearest ray-cube intersection, we can easily calculate a ray stop position for the SVO ray casting.

Fig. 1 presents rendering results of a virtual scene represented by a distance equation with a single SDF component.

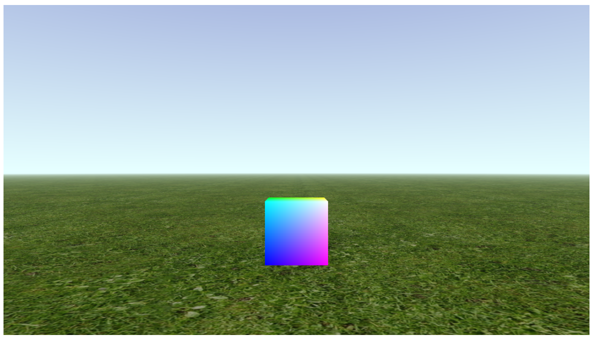

Figure 1: Ray marched SDF based virtual scene with a single SDF component. Rendered cube is shaded with calculated ray cast start position. 590 FPS with Nvidia GeForce GTX 660.

In order to add other elements to the scene, we can extend the global scene distance function by adding next SDF components to the virtual scene SDF equation.

For each component of the virtual scene equation, we have to calculate the value the component's SDF at the current iteration and process a minimum function per the ray marching iteration.

In the case of the virtual scene with three SDFs representing smaller and smaller objects in the scene, the distance equation can be expressed as follows:

$$\begin{aligned} d0 &= length(max(abs(p+o0)-b0),0) \\ d1 &= length(max(abs(p+o1)-b1),0) \\ d2 &= length(max(abs(p+o2)-b2),0) \\ d &= min(min(d0,d1),d2) \end{aligned} \tag{2}$$

where:

d	= distance to the object
$d0, d1, d2$	= distance to SDF component 0,1,2
p	= point on ray from the eye
$o0, o1, o2$	= SDF component offset from scene origin
$b0, b1, b2$	= scene SDF component 0,1,2 cube size

Fig. 2 presents rendering results of a virtual scene represented by a distance equation with three SDFs.

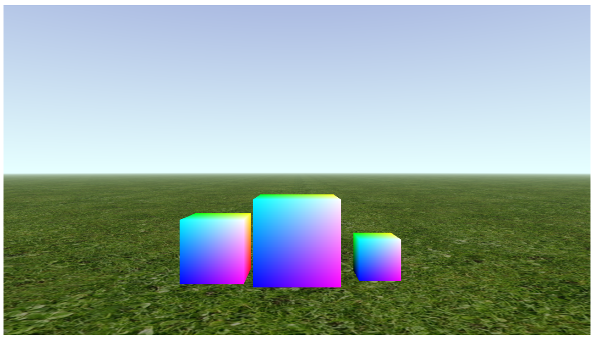
Figure 2: Ray marched SDF based virtual scene with the three SDF components. Rendered cubes are shaded with calculated ray cast start positions. 490 FPS with Nvidia GeForce GTX 660.

Obviously, the calculation of the unlimited number of minimum functions is impossible on the limited hardware. However, thanks to the simplicity and flexibility of the function-based scene representation, we are able, in theory, to handle the unlimited number of scene SDFs. It is possible because there is no need to store a large scene data in the GPU memory. That means, that the whole scene description of the highly complex virtual scene can be saved as a simple equation which makes the UOI rendering pipeline possible.

4.3 3D object visualization

Thanks to the use of distance functions as virtual scene equation components it is possible to create complex procedural objects. However, one of the main requirements of the developed rendering pipeline is the ability to visualize 3D objects created by artists. To achieve this goal, the cube primitives acquired from the SDF are replaced with the SVO-based 3D object. Moreover, we can use any available SVO algorithm for shading, object deformation and LOD management. In the implemented Unlimited Object Instancing rendering pipeline framework, we used a simple SVO ray casting.

Fig. 3 presents rendering results of the virtual scene represented by distance equation with the three SDF components with SVO based object ray casting.

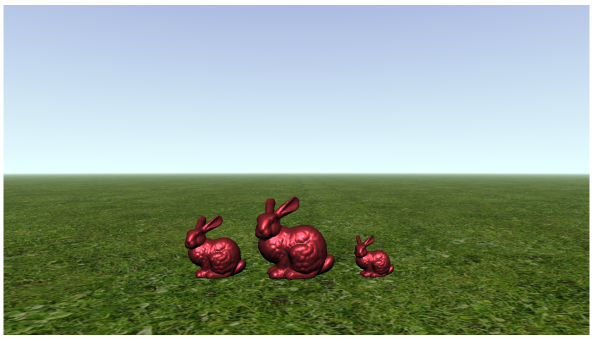
Figure 3: Ray marched SDF based virtual scene with the three SDF components integrated with SVO based 3D object ray casting. 220 FPS with Nvidia GeForce GTX 660.

The main problem related to the integration of SDF with SVO are potential occlusion errors. For SDF-based virtual scenes utilized in the "standard" way, ray-object intersection and occlusion errors would not happen.

If we want to replace SDF-based AABB boxes with SVO 3D objects, we may face the situation that a ray hits an AABB box but it misses an included SVO object. The details of our occlusion fixing solution are described in Sec. 5.

4.4 Global operator collection

The last component of our UOI rendering pipeline is the *Global operator* collection which is used to control the content and complexity of the scene in the spirit of the holistic paradigm. In order to create a complex scene by using a finite collection of the scene elements, global functions are applied to the SDF scene function. As mentioned before, one of the main features of the SDF functions is the possibility of multiplying objects by modifying primitives' distance functions.

The base operator that is used to obtain an unlimited number of instances per SDF component is the *Instancing Operator*. With the SDF-based object representation, using the modulo function it is possible to create objects in a controlled or unlimited way. The cube distance function 1 can be extended with *Instancing operator* as:

$$\begin{aligned} cell &= floor((p+size*0.5)/size) \\ p &= mod(p+size*0.5, size)-size \\ d &= length(max(abs(p)-b),0) \end{aligned} \tag{3}$$

where:

cell	= object instance grid cell
size	= repeat interval
d	= distance to the object
p	= point on the ray
b	= cube size

Instancing operator can be applied in any dimension, creating a 1D/2D/3D grid with the instanced cube objects. An important feature of the repetition function is that we acquire the cell id of every generated object. This information is then used by the all remaining operators as a unique input for the noise algorithms. In addition, we take advantage of the information to fix the occlusion errors. Fig. 4- 5 present rendering results of the virtual scene with applied *Instancing operator*.

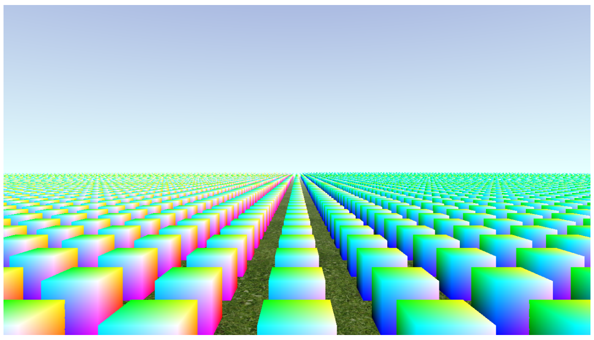

Figure 4: 2D *Instancing operator* applied for the single SDF component virtual scene. 215 FPS with Nvidia GeForce GTX 660.

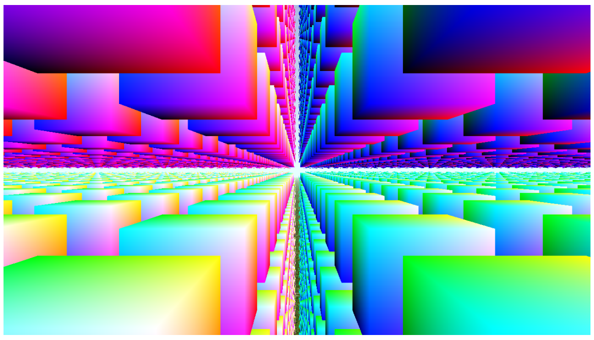

Figure 5: 3D *Instancing operator* applied for the single SDF component virtual scene. 160 FPS with Nvidia GeForce GTX 660.

As one can see in the figures 4 - 5 *Instancing operator* does not cause any noticeable performance loss regardless of the instancing dimension or a number of instances present in the scene.

Following the holistic approach, *Global operators* are applied to the whole scene. It means that we cannot control each object on the virtual scene independently.

Based on the operator's features, the developed *Global operators* have been classified into two groups.

4.4.1 Object operators

Object operators are used to control geometry and material data of the generated objects. Using the cell id acquired from the instancing operator as an input to the procedural noise algorithm (e.g. Perlin/Simplex noise), we can apply unique variations to generated objects. We have developed the following *Object operators*:

1. **Object type operator**—to choose an SVO data buffer which is used in the visualization algorithm.
2. **Material type operator**—to apply unique values for the objects material (e.g. albedo, roughness, metalness values).
3. **Existence operator**—the most advanced operator in the group. Using a noise function with the user input and features of the SDF representation, we can control the existence of generated objects. It is implemented by dynamically creating a new SDF element and performing the boolean subtraction from the virtual scene distance function. Also, if an SDF cube intersects another one it must be added to the final distance function with the boolean union operator. The same algorithm is used to fix the occlusion errors (see Sec. 5).

4.4.2 Transformation operators

The second group of the operators is used to apply an affine transformation to each generated object. We developed the following *Transformation operators*:

1. **Translation operator**—used to apply a translation transformation.
2. **Scale operator**—used to apply scale transformation.
3. **Rotation operator**—used to apply rotation transformation.

All *Transformation operators* are limited to the boundaries of the SDF component grid cell.

5 IMPLEMENTATION DETAILS

In this section, we describe important implementation details of our UOI rendering pipeline. We have implemented our method using OpenGL 4.5 API with C++14 but any other graphics interface or programming language can be used. All included shader source code listings are prepared in GLSL language. Due to the simplicity of SDF ray marching with sphere tracing, the presented approach can be easily implemented and integrated into all popular game engines. The only requirement is the support for programmable compute shaders.

5.1 Virtual scene visualization

The virtual scene visualization is directly based on classic SDF ray marching. The main extension is that in order to solve a potential occlusion error, ray marching is executed multiple times. Therefore, we need to take the previous hit distance into account and add it to the ray marching start position in the next iteration. Also, for an SVO-based 3D object rendering it is necessary to calculate the volume ray casting start and stop positions so as to perform SVO ray-casting. Listing 1 presents a simplified, SDF with SVO rendering pipeline compute shader code.

```
struct HitResult {
  vec3 cell;
  float distance;
  int lod;
  int material;
};

HitResult RayMarch(vec3 origin,vec3 direction) {
  HitResult hit;
  float distanceDelta = Near + OcclusionDelta;
  float travel = 0.0;
  vec3 position = origin;

  for(int i = 0; i < RayIterations; ++i) {
    position = position + direction * distanceDelta;
    hit = ObjectCube(position, size, lod, material);
    HitResult hit2 = ObjectCube(position, size2,
         lod2, material2);

    if(hit2.distance < hit.distance)
      hit = hit2;

    <Occlusion resolve>
    distanceDelta = hit.distance;
    travel += distanceDelta;

    if(travel > Far) {
      hit.distance = Far;
      return hit;
    }

    if (distanceDelta - Precision < 0.001) {
      hit.distance = travel;
      return hit;
    }
  }

  hit.distance = Far;
  return hit;
}

void main(void) {
  for (int k = 0; k < OcclusionIterations; ++k) {

  HitResult hit = RayMarch(position, direction);

  if (hit.material) // material < 0 means no hit
  {
    vec3 hitPosition = position + direction *
         hit.distance;
    float size = (1.0 / hit.size.x);

    vec3 rayStart = hitPosition * size;
    vec3 rayStop = GetRayStop(hitPosition,
         direction, size);
    bool missed = true;

    <Object LOD calculation>
    <ObjectRayCast>

    if (!missed) // save result
      return;
}
```

Listing 1: Unlimited Object Instancing rendering pipeline simplified visualization source code.

5.2 Occlussion error fixing

The biggest implementation challenge for the SDF and SVO rendering pipeline implementation is potential object occlusion errors. In order to tackle this problem, we need to execute multiple ray marching iterations, one for each occlusion error.

We developed two methods for occlusion fixing. Both of them are using an occlusion stack that is used to save a previous hit information—the grid cell 3D vector and the SDF component id.

The first method is a cell estimation. If we again hit the same object, we set a current distanceDelta to the defined *Escape* value which allows for omitting the current object.

However, this method will not work if two different SDF components intersect. Also, there is need to save the history of the occurred occlusions in the stack. In this case, we need to use the second, more universal method. Using the data from the previous occlusion error, we calculate the distance for the scene cell and, using the subtraction operator, we cut it from the scene SDF function. Listing 2 presents the source code for this method.

```
vec3 occlusionCell;
int occlusionID;

float interval = SceneIntervals[occlusionID];
vec3 offset = vec3(occlusionCell.x * interval,
     occlusionCell.y * interval,
occlusionCell.z * interval); vec3 q = (position +
     SceneOffsets[occlusionID]) - offset;

float occluder = ObjectCell(q,
     SceneSizesOccluder[occlusionID]);
hit.distance = max(-occluder, hit.distance);
```

Listing 2: Occlusion error fixing methods for SDF wih SVO rendering pipeline.

6 RENDERING AND PERFORMANCE TEST RESULTS

All depicted timings were obtained on Intel Core i5 2500K CPU with NVidia GeForce GTX 660 GPU and with NVidia GeForce GTX 980. All algorithms were implemented using OpenGL 4.5 API with C++14 for Windows 10 64-bit. We used Stanford Repository models as a test object [Stanford11].

The presented rendering results show that the developed rendering pipeline is efficient and offers real-time performance even for a medium power hardware. Unfortunately, the results also show that the SDF based virtual scene representation suffers for a visible regularity of the object distribution. Also, SDF ray marching seems to be slower than for example some sort of the uniform grid traversal algorithm like 3D-DDA.

Using instancing operator we create an infinite uniform grid with defined cell size and interval. The obvious

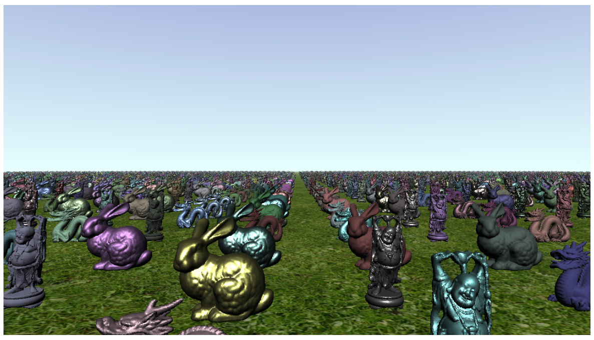

Figure 6: The virtual scene with 2D instancing operator applied. 50 FPS on Nvidia GeForce GTX 660, 183 FPS on Nvidia GeForce GTX 980.

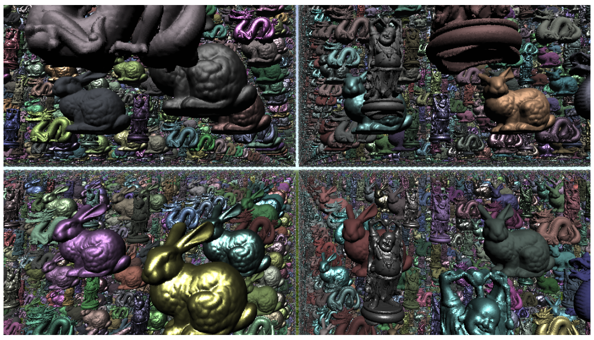

Figure 7: The virtual scene with 3D instancing operator applied. 20 FPS on Nvidia GeForce GTX 660, 67 FPS on Nvidia GeForce GTX 980.

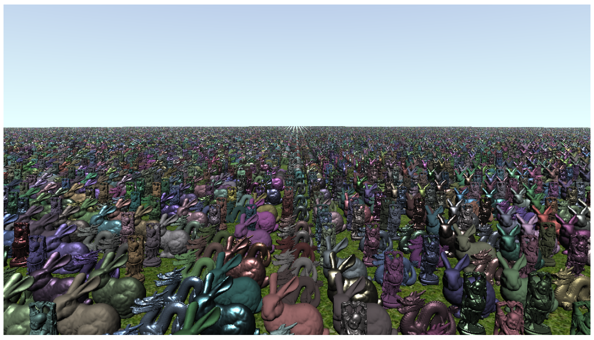

Figure 8: The virtual scene with instancing, type and material operators applied. 35 FPS on Nvidia GeForce GTX 660, 98 FPS on Nvidia GeForce GTX 980.

alternative seems to be a procedural definition of an infinite uniform grid along with a variation of the Bresenham algorithm for the grid traversal. Such a method would be compatible with the remaining components of the holistic approach and could be applied for, e.g., figures 8 - 11. It would presumably offer better performance results thanks to the simplicity of the uniform grid traversal. However, for the rest of the presented figures SDF it could be not applied.

First, with SDF we can efficiently use many independent SDF components to represent the virtual scene. It means that the virtual scene may contain N uniform grids with different attributes and the possible intersection between their cells. Such a virtual scene is presented in the figures 12 - 15. For virtual scenes like those, we could not use the traditional uniform grid traversal algorithm. Moreover, we are not limited only to SVO based 3D objects. Using SDF ray marching we could render highly complex 3D objects and calculate all necessary data for realistic shading like normal vectors, ambient occlusion or even soft shadows. For these reasons, SDF based virtual scene representation seems to be a better solution than the procedural uniform grid for Unlimited Object Instancing rendering pipeline.

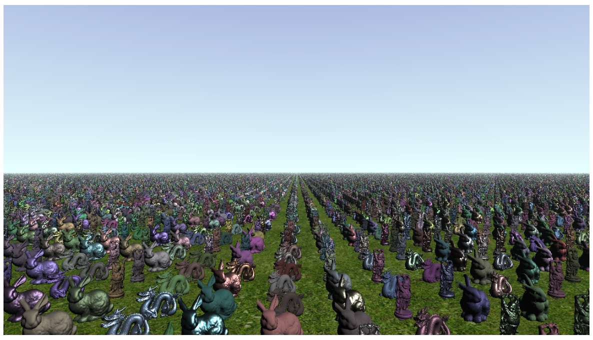

Figure 9: The virtual scene with translation and rotation operators applied. 27 FPS on Nvidia GeForce GTX 660, 81 FPS on Nvidia GeForce GTX 980.

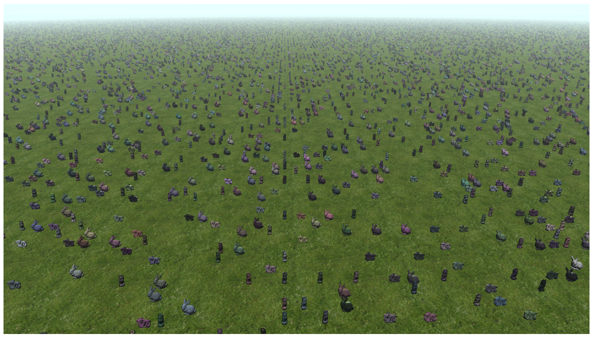

Figure 10: The virtual scene with existence operators applied. 31 FPS on Nvidia GeForce GTX 660, 80 FPS on Nvidia GeForce GTX 980.

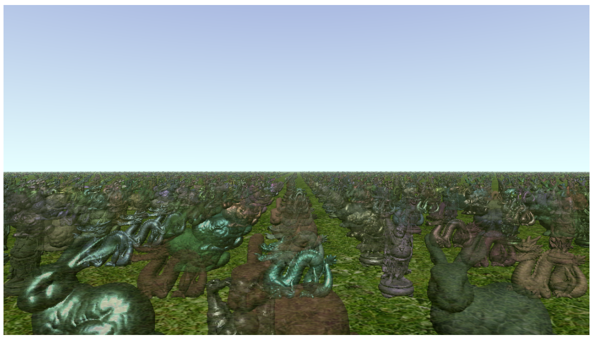

Figure 11: The virtual scene with translucent objects. 34 FPS on Nvidia GeForce GTX 660, 55 FPS on Nvidia GeForce GTX 980.

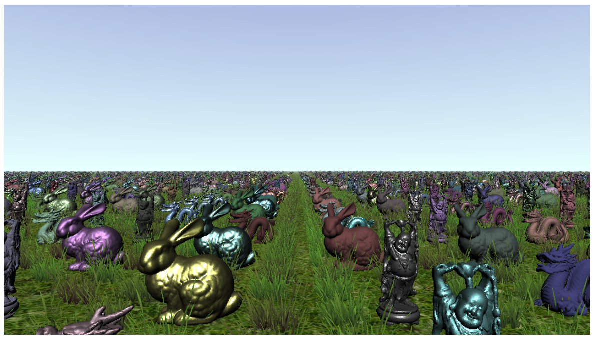
Figure 12: The virtual scene with two SDF components - one for Stanford objects and one for the grass objects. 24 FPS on Nvidia GeForce GTX 660, 100 FPS on Nvidia GeForce GTX 980.

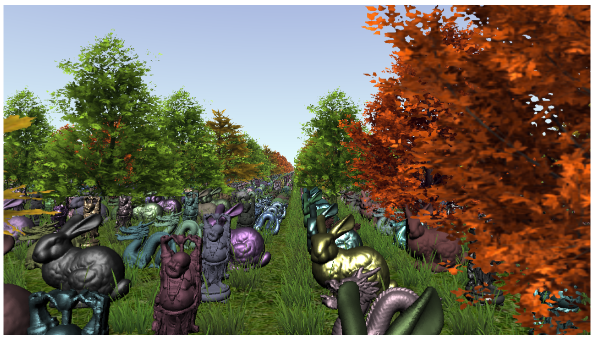
Figure 13: The virtual scene with three SDF components - one for Stanford objects, one for the grass and one for the trees. 32 FPS on Nvidia GeForce GTX 660, 93 FPS on Nvidia GeForce GTX 980.

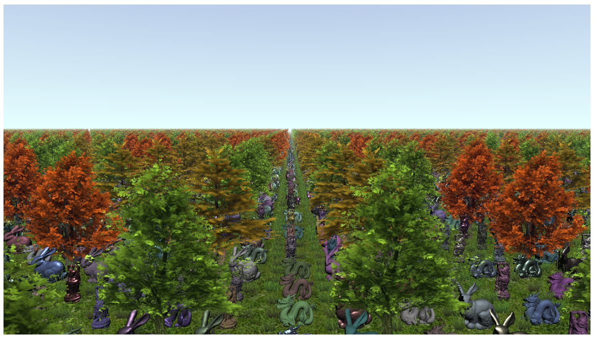
Figure 14: The virtual scene with three SDF components - one for Stanford objects, one for the grass and one for the trees. 23 FPS on Nvidia GeForce GTX 660, 57 FPS on Nvidia GeForce GTX 980.

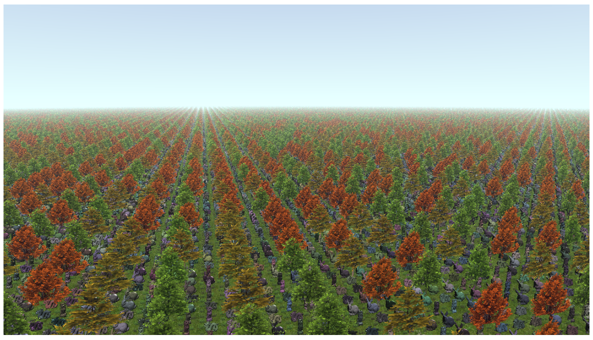
Figure 15: The virtual scene with three SDF components - one for Stanford objects, one for the grass and one for the trees. 29 FPS on Nvidia GeForce GTX 660, 65 FPS on Nvidia GeForce GTX 980.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we presented a novel approach to efficient rendering of an unlimited number of 3D objects in real-time. Thanks to the newly proposed computer graphics paradigm—the *Holistic Graphics Programming*, we created rendering pipeline that can process as many unique instances of 3D objects as we want in real-time. Using a Signed Distance Function, we limited memory requirements for the virtual scene description, making processing an unlimited number of the 3D object for each SDF component possible. Moreover, taking advantage of a collection of developed *Global operators* we are able to control the content and the complexity of the virtual scene in a procedural way.

Thanks to the Sparse Voxel Octrees integrated with the SDF representation, we are able to render high-resolution 3D objects created by artists. In order to integrate the SDF-based scene with SVO-based objects, we developed an occlusion fixing algorithm. Finally, the developed single pass rendering pipeline can be easily integrated with the e.g. triangle rasterization pipeline for animated, user-controlled objects.

An obvious step forward would be an implementation of *Global operators* that can be used to create dynamic scenes. A good idea seems to be the usage of the dynamic sparse textures to control objects' movement and the existence of the SDF components on the virtual scene. Also, a further optimization and extension for the SDF ray marching rendering pipeline should be considered. For example the distance field based soft shadow should be easy to implement to increase the depth and immersion of the virtual scene.

8 REFERENCES

[Bau11] Bautembach, D., Animated sparse voxel octrees, Bachelor Thesis, University of Hamburg, 2011.

[Cao10] Cao, X., Zhang, Y., Gao, S., Teng, R., Wang, X., The design and implement of Scene Management in 3D engine SR, 2010 International Conference on Mechanic Automation and Control Engineering, pages 183-186.

[Crassin11] Crassin, C., Neyret, F., Sainz, M., Green, S., and Eisemann, E., Interactive indirect illumination using voxel cone tracing, Computer Graphics Forum (Proceedings of Pacific Graphics 2011), vol. 30, no. 7, sep 2011.

[Deussen98] Deussen, O., Hanrahan, P., Lintermann, B., Mesh, R., Pharr, M., Prusinkiewicz, P., Realistic modeling and rendering of plant ecosystems, Proceedings of SIGGRAPH 98, Orlando, Florida, July 19-24, 1998, In Computer Graphics Proceedings, Annual Conference Series, 1998, ACM SIGGRAPH, pages 275-286.

[Domaradzki16] Domaradzki, J., Martyn, T., Fracturing Sparse-Voxel-Octree objects using dynamical Voronoi patterns, Computer Graphics, Visualization and Computer Vision WSCG 2016. Full Papers Proceedings / Pan Zhigeng, Skala Vaclav (red.), Computer Science Research Notes, vol. 2601, 2016, Vaclav Skala - UNION Agency, ISBN 978-80-86943-57-2, pages 37-46.

[Green07] Green, C., Improved alpha-tested magnification for vector textures and special effects, Proceeding SIGGRAPH '07 ACM SIGGRAPH 2007 courses, pages 9-18.

[Hart89] Hart, J., C., Sandin, D., J., Kaufmann, L., H., Ray Tracing Deterministic 3-D Fractals Computer Graphics 23(3), (Proc. SIGGRAPH 89,) July 1989, pages 289-296.

[Hart91] Hart, J., C., DeFanti, T., A., Efficient antialiased rendering of 3D linear fractals. Computer Graphics (SIGGRAPH 91 Proceedings), 25:91-100, 1991.

[Hart92] Hart, J., C., The object instancing paradigm for linear fractal modeling. In Proceedings of Graphics Interface 92, pages 224-231, 1992.

[Hart94] Hart, J., C., Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces, The Visual Computer, Volume 12, pages 527-545.

[Hart97] Hart, J., C., Implicit Representations of Rough Surfaces Computer Graphics forum, Volume 16, Issue 2, June 1997, pages 91-99

[Jab16] Jabłoński, Sz., Martyn, T., Real-Time Rendering of Continuous Levels of Detail for Sparse Voxel Octrees, Computer Graphics, Visualization and Computer Vision WSCG 2016. Short Papers Proceedings / Skala Vaclav (red.), Computer Science Research Notes, vol. 2602, 2016, Vaclav Skala - UNION Agency, ISBN 978-80-86943-58-9, pages 79-88.

[Kampe13] Kämpe, V., Sintorn, E., Assarsson, Ul, High Resolution Sparse Voxel DAGs, ACM Trans. Graph., vol. 32, no. 4, pages 1-13.

[Kay86] Kay, T., L., Kajiya, J., T., Ray tracing complex scenes, Computer Graphics, SIGGRAPH 86 Proceedings, 20(4):269-278, 1986.

[Keinert14] Keinert, B., Schäfer, H., Korndörfer, J., Ganse, U., Stamminger, M., Enhanced Sphere Tracing, STAG: Smart Tools and Apps for Graphics, 2014, pages 1-8.

[Laine10] Laine, S., and Karras, T., Efficient sparse voxel octrees, in Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, ser. I3D 2010. New York, NY, USA: ACM, 2010, pages 55-63.

[Lueb02] Luebke D., Watson B., Cohen, J., D., Reddy, M., and Varshney, A., Level of Detail for 3D Graphics, New York, NY, USA: Elsevier Science Inc., 2002.

[Martyn10] Martyn T., Chaos and graphics: Realistic rendering 3d ifs fractals in real-time with graphics accelerators, Comput. Graph., vol. 34, no. 2, pages 167-175, Apr. 2010.

[Reiner11] Reiner, T., Mückl, G., Dachsbacher, C., Interactive modeling of implicit surfaces using a direct visualization approach with signed distance functions, Computers and Graphics, Volume 35 Issue 3, June, 2011, pages 596-603.

[Stanford11] The Stanford 3D Scanning Repository, Stanford University, 22 Dec 2010, Retrieved 17 July 2011.

[Suther63] Sutherland, I., E., Sketchpad: A man-machine graphical communication system, Proceedings of the Spring Joint Computer Conference, 1963.

[Snyder87] Snyder, J., M., Barr, A., H., Ray tracing complex models containing surface tessellations, Computer Graphics SIGGRAPH 87 Proceedings, 21(4):119-128, 1987.

[Wil13] Willcocks, C. G., Sparse volumetric deformation, Ph.D. dissertation, Durham University, 2013.

[Wright15] Dynamic Occlusion with Signed Distance Fields, Advances in Real-Time Rendering in Games, SIGGRAPH 2015.