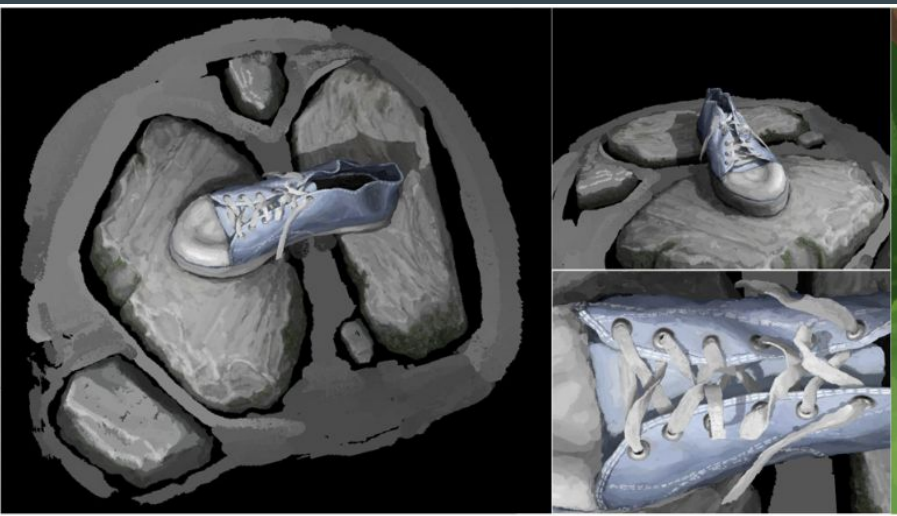
Dynamic Deep Octree for High-resolution Volumetric Painting in Virtual Reality

Authors - Yeojin Kim, Byungmoon Kim, Young J. Kim
Presented By - Anshul Mendiratta

Introduction

AIM

- To digitally paint on a 3D voxel canvas in virtual reality similar to how one would paint on a 2D pixel canvas.
- Use a dynamic octree based painting and rendering system using both CPU and GPU resulting in low latency without compromising frame rates
- Canvas size of 40 km^3 with details down to 0.3mm^3



(a) A Sneaker

(b) Spring Concert

Challenges

• **Dynamic tree update -** Artists will continuously modify the underlying tree; we therefore need to update the tree dynamically.

• **Constant frame rates -** Artists spend several hours painting in VR, and consideration must therefore be given to mitigating the possibility of VR-sickness. One source of such sickness is hitching or stuttering in the rendering frame rates, which should not be compromised; the frame rates should stay constant.

• **Low-latency stroke display -** When an artist applies a stroke, the tree should be modified immediately and rendered back to the artist. Therefore, we require low latency for stroke display.

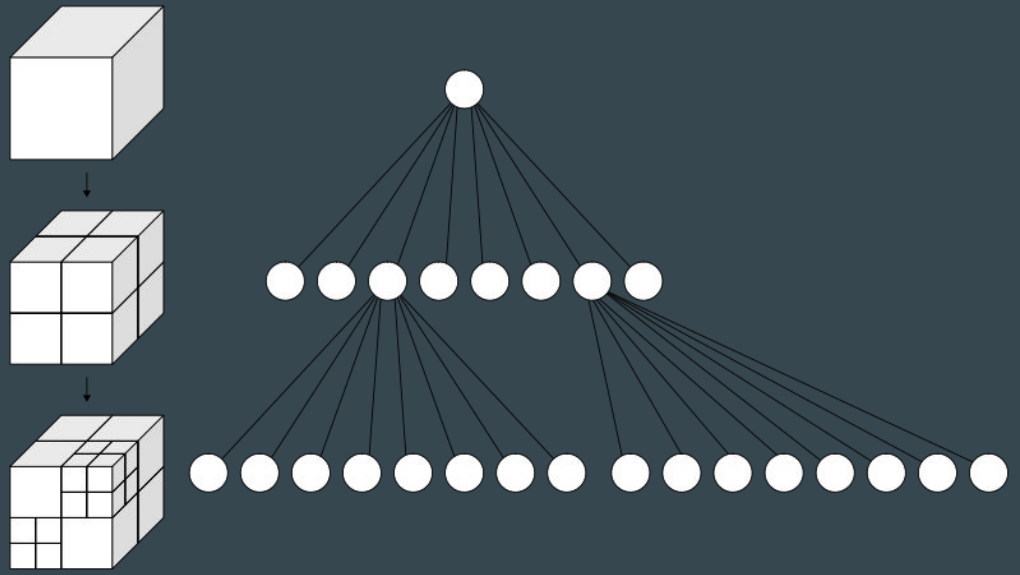
• **Low memory consumption -** When a very large canvas is used, artists tend to paint a very large world and add details in multiple locations. Thus, a low memory requirement is beneficial for maintaining a large canvas.

Octree: A recap

An octree is a tree data structure in which each internal node has exactly eight children. Octrees are most often used to partition a three-dimensional space by recursively subdividing it into eight octants.

The Octree can be formed from 3D volume by doing the following steps:

1. Divide the current 3D volume into eight boxes
2. If any box has more than one point then divide it further into boxes
3. Do not divide the box which has one or zero points in it
4. Do this process repeatedly until all the boxes contain one or zero point in it



Time complexity:

1. Find: O(log2N)
2. Insert: O(log2N)
3. Search: O(log2N)
4. Space complexity: O(k log2N)

Where k is count of points in the space and space is of dimension N x M x O, N >= M, O.

Overview

Overview of the System

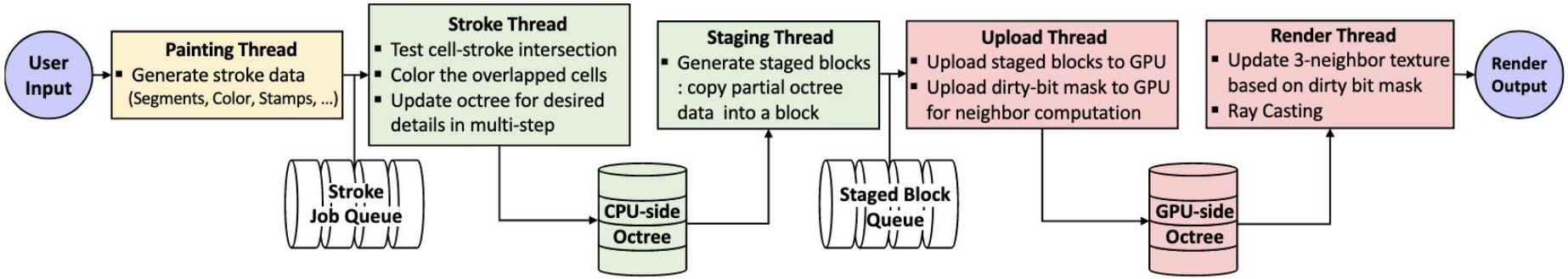


Figure 4: *A sequence of CPU-side threads from user input to rendering output. After applying brush strokes, the octree is dynamically adjusted and incrementally uploaded to GPU for rendering. We enable uninterrupted painting by using the stroke job queue, and enable uninterrupted rendering by staging blocks. We achieved very small latency from painting to rendering, measured at under 20 milliseconds on average.*

Overview of the System

Interactive volumetric field painting is composed of several parallel tasks: processing strokes, adjusting the octree, uploading the octree to the GPU, and rendering the octree. These are implemented in multiple threads as illustrated in the previous figure.

1. In the painting thread, the segments, color, and the stamp of strokes are queued.
2. In the stroke thread, we conduct a stroke-cell intersection test, and refine or coarsen the intersecting cells. The octree memory is divided into a uniformly-sized blocks.
3. In the staging thread, a sequence of cells is copied including newly refined or coarsened cells into a separate memory, which is called the staged block. These staged blocks are pushed to the staged blocks queue.
4. In the upload thread, this staged queue is consumed by uploading staged blocks to the GPU.
5. Finally, the rendering thread renders the octree using ray casting.

Memory Layout

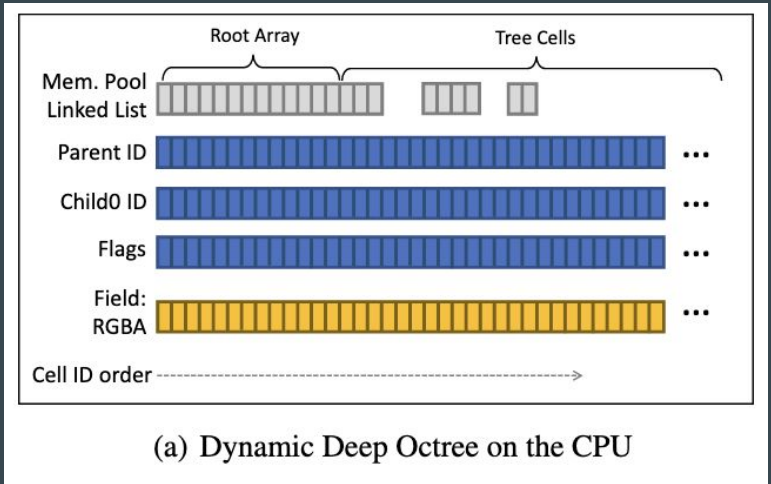
Root array and octree depth

- In order to author highly-detailed and dynamic volumetric fields, an array of octrees is used.
- The roots of the octree are stored as a 3D array, called the root array. Each root can be refined up to 24 times, which is the maximum depth.
- While the root array helps to reduce the tree depth and offers several advantages such as trivial parallelization, in a very large canvas with highly adaptive tree, the advantage appears to diminish.
- Therefore, a relatively coarse, 4^3 array of octree roots, each of which can be refined to a maximum depth. Effectively, this is equivalent to a single 26-deep octree root. This resolution can span a volumetric space of from $0.3mm^3$ to $40Km^3$ with respect to the room-scale VR setup.

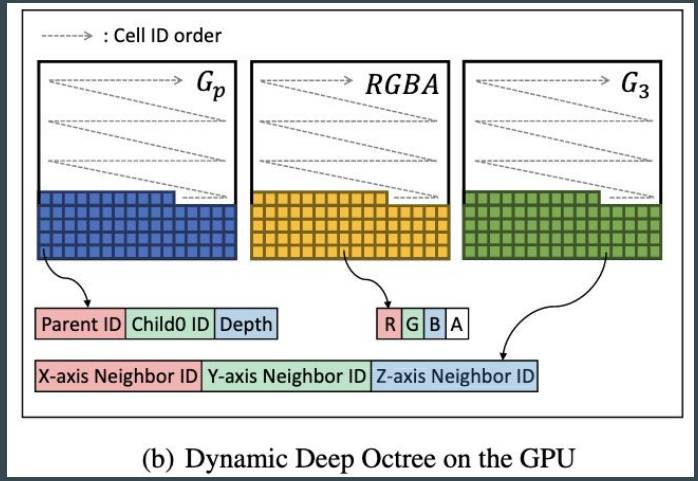
Octree on CPU vs Octree on GPU

In general, the CPU-side octree is constructed of linear pools (parent/child indices and fields) that are packed into textures in the GPU. The CPU has temporary pools (for painting) that do not exist in the GPU. For an efficient ray traversal, the GPU has a neighborhood connectivity pool, G3 texture, that CPU does not have.

Therefore, there is A CPU-side octree for dynamic adjustment and color blending, pick up, erase, and recolor operations and a GPU-side octree for rendering.



(a) Dynamic Deep Octree on the CPU



(b) Dynamic Deep Octree on the GPU

Dynamic Deep Octree Representation in a CPU

1. The octree is 2:1 balanced, where the difference in depth between neighbouring cells is equal to or less than 1
2. Eight children are created when a cell is refined.
3. Pointers not used, instead the index used that uniquely identifies each cell.
4. The octree is made up of multiple linear memory pools indexed by I. Painting properties such as color, density, and temporary variables are stored as separate field pools. More field pools can be added, even dynamically if needed, such as alpha values.
5. For dynamic refinement and coarsening, linked-list based memory management is used.
6. The size of an allocation unit is fixed as we allocate or free eight cells.
7. The pool begins with a uniform root array, i.e., 4^3 array, that cannot be freed. We have another separate flag pool for the depth of cells and other bit-field flags for tree adjustment.

Tree Graph and Neighborhood access

1. Consider tree graph $G_{p,}$ such that the octree is defined only by parents and children pools that store two indices of the parent and the first child, since indices of the remaining seven children are consecutively numbered and hence do not need to be stored.
2. Note that ray casting using only Gp may have poor performance as the depth of the tree increases; e.g. if the maximum depth is 24, the traversal path from Gp to a neighbor can be as long as 48 in the worst case (Cells belonging to different root nodes). Therefore, an immediate neighbor topology is useful to accelerate the ray traversal.
3. Since we have a 2:1 balanced octree, each cell can have 6 - 24 neighbours

Tree Graph and Neighborhood access

1. First we link same or smaller depth cells. Hence we have 6 neighbours per cell.
2. Now, since eight children have consecutive indices, we can access at least 3 of the 6 neighbours directly from their index. For example D_1, D_2 and the cell above or below D.
3. The other 3 cells may result in long tree traversals, hence they are precomputed . The collection of 3 neighbours for each cell is termed as G_3
4. To access any of the subcells in a case like D_1, we can again make the use of indices and the fact that we have access to the first child of every cell

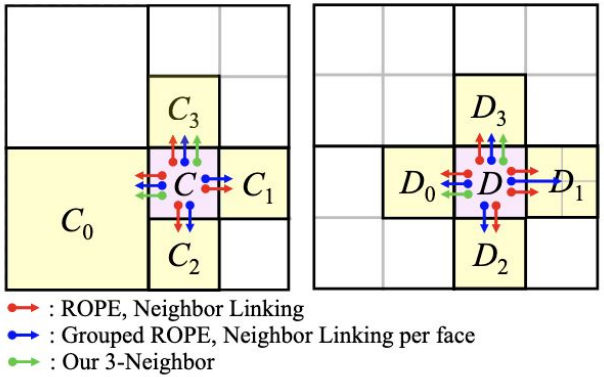


Figure 6: *Neighbors of C (left), and D (right) in the quadtree. C, C_1, and C_2 share the same parent, and hence computing C_1 and C_2 from C is easy. On the other hand, tree distances between C and C_0, C_3, and between D and D_0, D_3 can be very large. Therefore, we precompute C_0, C_3, D_0, and D_3 and stored them in a separate texture.*

Dynamic Deep Octree Representation in a GPU

- Mapping the octree pools in a CPU to textures in a GPU is straightforward.
- Since textures have a resolution-limit in each dimension, we cannot use a 1D texture.
- We must use 2D or 3D textures and map a linear index I to two or three indices. Since modern GPUs support up to 16 thousand texels per dimension, 2D textures can support up to 256M cells.
- We pack Gp (parent, child) and the depth into a texture. RGBA color is stored as another texture. G3 (3-neighbor) is another integer texture with three channels.

Updating the Octree

Dynamic Octree Update with low latency

1. The importance of avoiding nausea, sickness, and postural instability is escalated in VR painting lasting several hours.
2. Among factors on those symptoms, little delay in sync between the rendered scene and the head motion is an minimum requirement that cannot be compromised.
3. However, simply copying an octree to a texture will take 222ms even with full bandwidths of a CPU, a PCIe, and a GPU.
4. Therefore, instead of updating the entire tree, we incrementally and dynamically update the underlying octree.

Incremental Tree Adjustment

1. Although we can effectively reject cells that do not intersect with brush stamps, painting an octree can still be expensive.
2. A broad brush stroke can be applied near a highly-refine region or a fine brush stroke can be applied near a coarsened region.
3. To address a sharp change in the depth of cells, we develop a multi-step strategy.
4. In a CPU-thread separate from rendering, we first paint on the CPU tree without tree adjustment and update on the GPU.
5. The next step is a tree-adjustment stage where we mark cells that should be refined or coarsened and perform one-level refinement or coarsening per frame. After one-level tree adjustment, we reflect these changes to the GPU.
6. We repeat this process until no cell needs to be refined or coarsened.

Block-based Update using Staging

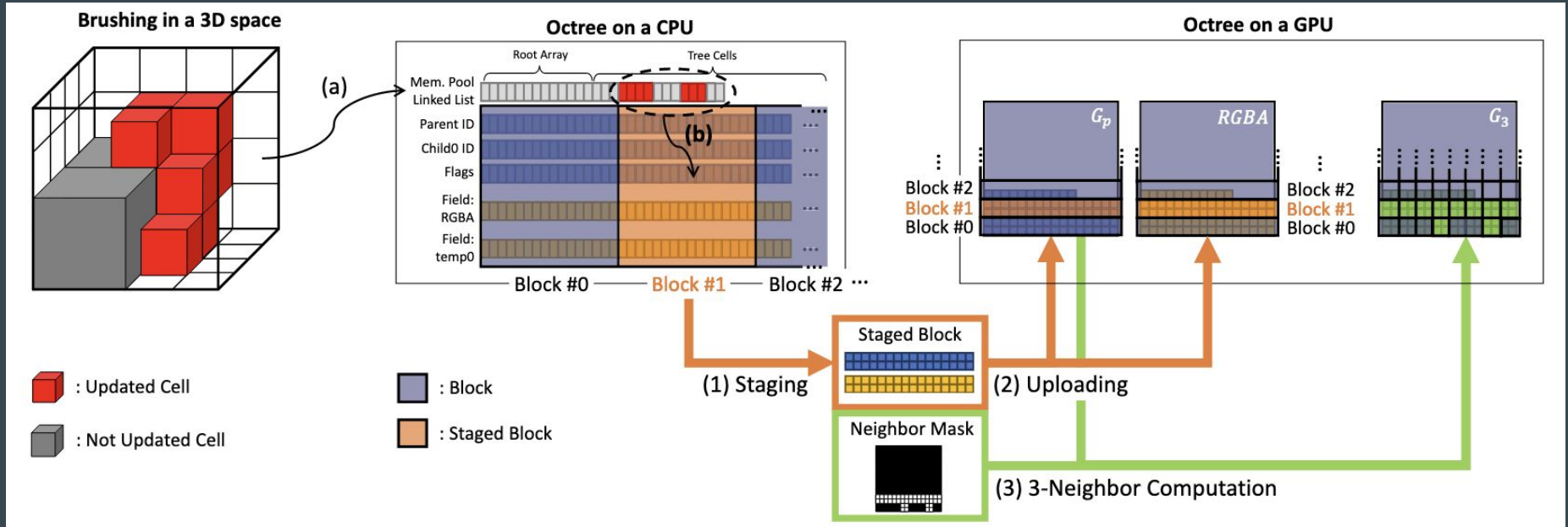


Figure 7: *Staged updates with a neighbor computation mask. (a) Brushing a stroke in a 3D space is local not only in the space but also in the memory (red cells in the 3D space and the octree on a CPU). Therefore, we divide the octree into blocks shown as sky blue and orange boxes. (b) Among blocks, we find a block (orange box) containing updated cells (dotted ellipse). In the middle of the painting, (1) the block on the CPU is staged with a neighbor computation mask and (2) uploaded to the GPU. (3) Note that $\mathcal{G}_3$ is not uploaded, but is rather computed on the GPU.*

Block-based Update using Staging

1. The stroke diameter is set to be approximately 10 cells, for a stroke length of one, about up to 1,000 cells per stroke would require updates.
2. Since uploading a 1,000 times to the GPU would also be prohibitively slow, we use large blocks to reduce the upload counts.
3. We use a block, which refers to linear pitched packing that divides texture horizontally.
4. If we directly upload updated blocks to the GPU, the whole CPU-based tree would be locked and the painting thread would stall.
5. To avoid this painting interruption, we first copy the block to a staging buffer, designed for CPU-side hazard control, that serves as an update queue. A block copied to a staging buffer is called a staged block.
6. We collect the staged blocks in a separate thread using only small atomic sections during tree adjustments and queue them in the LIFO queues with upload-to-GPU tasks.
7. We then simply stage and upload one block per rendering frame

Neighborhood Computation Mask

1. Even though a neighbor computation on the GPU corresponds to a simple computation of the graph G3, the resolution of the texture can be large (16384×8192) which affects the interactivity.
2. Therefore, we develop a simple and efficient method to dramatically reduce this rendering cost.
3. Once a cell is created or deleted, not only the cell but also its neighbors should be updated in G3. Since neighbors may not be inside a block that contains the cell, updating G3 within the block would not be sufficient.
4. Hence, we compute a very small mask in the CPU that contains dirty bits indicating which cells need to recompute their neighbors due to the tree topology change. While staging the cells on the CPU, we upload this small mask to the GPU, and perform a neighbor computation only on the cells in the marked area.

Rendering the Volume

Accurate Volume Rendering with High-depth Octree

1. Rendering 3D volumetric fields poses another challenge.
2. One viable solution involves extracting voxel faces and rendering them through raster graphics pipeline using, for example, OpenGL
3. However, as the number of grids in the non-uniform size increases, the extracted vertex positions, particularly far from the origin, may not be accurate due to numerical error.
4. More significantly, geometric extraction requires a substantial amount of computational time, as the number of voxels grows.
5. Consequently, we explore an alternative approach of ray casting through octree volumetric field

Accurate ray starting point

1. When editing fine detail, users should be able to zoom in to observe the cells that have the highest depth.
2. However, the size of these tree cells can be even smaller than the single-precision floating point.
3. In a VR environment, naively using the floating point for the eye position in a world coordinate will force the head positions to jump towards nearby floating point values, and more significantly, the eye distance will be erratic.
4. We propose computing the ray starting point in a cell local coordinate frame. Our cell-local coordinate system is a barycentric coordinate system that has the origin at the cell center with a size of one. The range of coordinates inside the cell is [-0.5, 0.5]. Consequently, the finest resolution inside a cell is 0.5×2^{-23} in single-precision floating point regardless of the size of the cell.

Accurate ray traversal

1. Given a ray and its direction d, and a cell-entry point pi of the ray into the ith cell, we compute the cell traversal distance ti and the cell-exit point pi′ as well as a neighboring cell containing pi′.
2. Since our volumetric canvas covers a large space and the cell sizes vary by a large magnitude, using a global coordinate system to calculate pi′ and the ray traversal length t can be inaccurate.
3. In contrast, the cell-local coordinate system can produce accurate results regardless of the zoom level. We represent pi and p′i with respect to the frame whose origin is located at the cell center and the size is normalized to one.
4. Using the intersecting face which contains pi′ and the neighbor texture described, we choose the neighbor cell (the i + 1th cell) to visit, and set pi′ to pi+1′. This process is repeated until the ray terminates after accumulating full opacity or exits the canvas.

Results

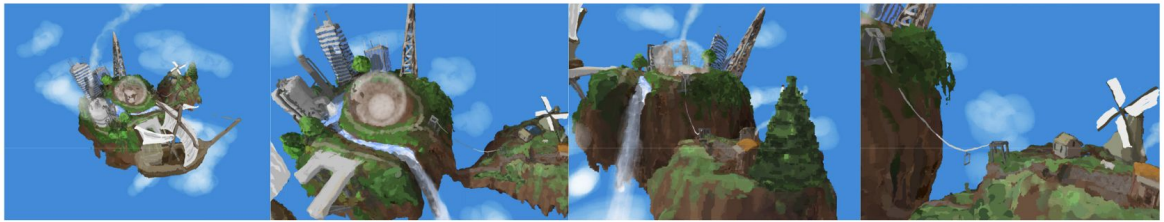


Figure 13: *"Island" from different view points.*

Figure 14: *"Nature" from different view points.*

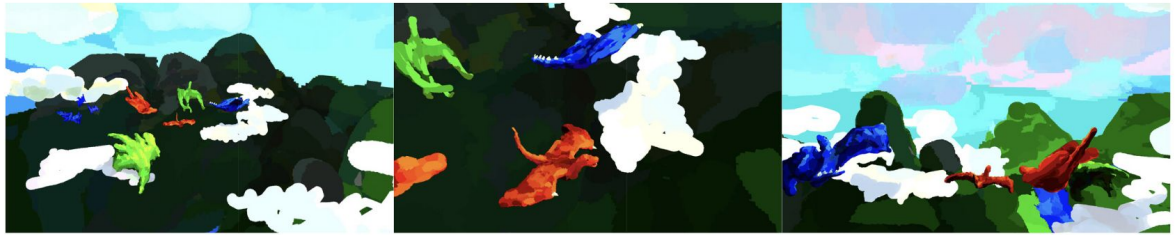


Figure 15: *"Flying dragons" from different view points.*