Tiling Layered Depth Images

Jonathan Shade Michael F. Cohen* Don P. Mitchell*

University of Washington *Microsoft Research

Abstract

We present a system for modeling and real-time rendering of solid terrains. Using results from the field of tiling, we show how a global 3D texture can be computed for a 2D terrain using a small set of Layered Depth Images. We propose a new set of Wang tiles that are shown empirically to tile the plane without apparent periodic structure. Furthmore, we introduce a new image-based data struture: multiresolution view-dependent Layered Depth Images. As our results show, this method produces natural looking 3D textures, with full parallax, in real time.

Keywords: tiling, image-based rendering

1 Introduction

Modeling and rendering scenes that capture the complexity of the real world is not an easy task. The manual effort required to model natural environments and the computational cost required to render such scenes are both very high. To date, systems that attempt to render realistic looking natural environments have solved either one problem, or the other, but not both.

Real-time efforts to this end typically rely on texture-mapped terrains combined with billboarded trees and bushes. Unfortunately, the result is, in essence, nice looking Astroturf. Examples include: much of the work done in flight simulators over the past thirty years. Flight simulations introduced billboarding: drawing approximations of objects (typically trees) using alpha matted polygons. A recent example is EverQuest[22], an on-line role-playing fantasy game. EverQuest provides a first-person 3D view of a medieval virtual world, but due to the limitation of current consumer graphics cards it uses a very simple texture mapped terrain.

At the other end of the spectrum, a lot of effort has been devoted to realistically modeling plants and terrain. Animatek's World Builder [3] and Bryce from Meta Creations [5] are two commercial software packages that can model and render realistic looking outdoor scenes. The output from both packages is an image and, thus, the user cannot view the scene interactively. Modeling realistic looking plants has been the focus of Przemyslaw Prusinkiewicz and his graduate students for many years. Radomír Mech [13] has recently shown extremely detailed and natural looking models of trees. These models are so complex, however, that rendering them in real-time is not possible. Oliver Deussen et al. [6] have demonstrated a system for modeling and rendering outdoor scenes, albeit also at great computational cost.

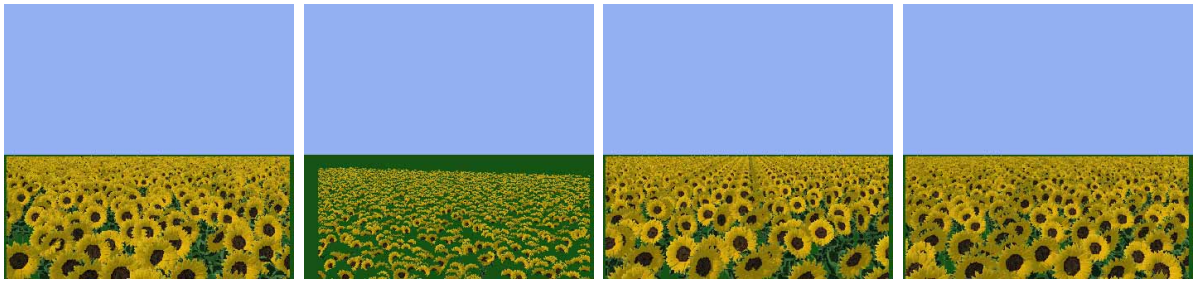
The goal of the project presented in this paper is to bridge these two areas of research. More specifically, we intend to create an interactive rendering of a terrain with a solid texture representing the vegetation. The tools we have chosen to pursue our goal are Layered Depth Images (LDIs) [19] combined with two-dimensional tilings of the plane. By computing a small set of complex LDIs and properly stitching them together in a non-periodic pattern, we can show that a natural looking environment can be rendered in real-time. Our solution is applicable to any interactive 3D environment that uses a terrain. Games and virtual worlds are the obvious beneficiaries of this work. However, our technique of using a three-dimensional tile in a two-dimensional tiling is one that can be widely applied in the field of computer graphics. There are many problems for which a 2.5D approximation to a 3D problem is suitable.

Figure 1: If Yosemite valley was covered in sunflowers.

1.1 Related work

Kajiya and Kay [10] built a system for rendering fur that used deformed volumes to represent a volumetric texture. Their system is akin to ours in that they deformed the volumes to get local variation. However their systems was not interactive, it was rendered using a raytracing algortihm. Neyret [15] extended Kajiya's work to use multiresolution volumes as a technique for antialiasing the animation of raytraced volumetric textures. In later work, Meyer and Neyret [14] showed how to use graphics hardware to accelerate volume textures tiled on a surface. The tiles they used had toroidal edge constraints (north matches south and east matches west). So, their tiling is inherently periodic. To add variation to the texture they deform the volume elements according to a height field mapped over an object.

Aperioidic texture mapping of surfaces was the subject of Neyret and Cani [16]. In this paper, they show how to tile a surface aperioidically using triangluar tiles. They map two dimensional triangluar textures onto their surfaces. Using the techniques we present, it may be possible to extrude their triangluar tiles to model solid textures over an object. Jos Stam explored using Robinson's set of 16 Wang tiles to texture map the plane aperiodically [21]. Stam's tilings were small, 6x8, so the structure apparent in the Wang 16 tilings seen in Figure 4(d) was not visible in his images. The stochastic tile set introduced in this paper would enhance



(a) (b) (c) (d)

Figure 2: (a) A Layered Depth Image of the scene that inspired this work, (b) the same LDI from a different view, (c) modeling the scene using a single tile, (d) modeling the scene using the 8 Wang tiles introduced in this paper.

Stam's work, allowing large non-periodic tilings to be computed easily.

There is a large body of work dealing with the problem of accelerating the display of very complex scenes [18, 20, 1, 2]. These systems are typically geared toward optimizing the use of polygon rendering hardware used in conjunction with image caching. Weber and Penn [25] developed a method for multiresolution modeling of realistic looking trees. These models were employed in a system that generated images of realistic looking terrains, although not in real time. Lastly, Deussen [6], recently presented a system that uses approximate instancing to model expansive natural looking scenes. Approximate instancing is similar to tiling in that it attempts to make non-perioidic imagery by repeating a small set of prototypical object throughout the scene.

2 Tiling

The inspiration for this work was the image in Figure 2(a). This scene consists of about seven thousand randomly placed sunflowers. There are only eleven unqiue sunflower models; instancing is employed to avoid modeling each flower individually. We first constructed a standard Layered Depth Image of the scene. This provides a very satisfying result and allows camera motion near the LDI center but breaks down quickly away from this point, as seen in Figure 2(b).

Our first attempt at modeling this scene using tiles was to use just one tile. A tile in this case consists of a square plot of terrain with about 20 sunflowers in it. This type of tile has toroidal edge constraints: the north side matches the south side and the east side matches the west side. Figure 11, at the end of the paper shows sets of tiles similar to this one. Figure 2(c) shows a rendering of the tiled scene. There is a pronounced "corn row" effect in the image, and the periodicity in the tiling is obvious. After this initial failure, we turned to the study of tilings to help us create tiled sunflowers that would look as natural as Figure 2(a).

A *tiling of the plane* is a countable family of tiles $\mathcal{T} = \{T_1, T_2, \ldots\}$ which cover the plane without gaps or overlaps [8]. In other words, every point on the plane must be a member of some T_i for some i, and the intersection of any two tiles must be empty. Tiles can take many different shapes, from triangles to squares, to complex polygons. In this paper we use a simple class of square tiles called Wang tiles [23, 24].

A recent development in the theory of tilings has demonstrated the existance of sets of prototiles which admit infinitely many tilings the plane, none of which are periodic. The first such set of prototiles discovered was comprised of tiles known as the Wang tiles. Wang tiles are square tiles with colored edges. The edges of any two adjacent tiles in a tiling must match, and the tiling must consist only of translations of the prototiles, rotations and reflections are not allowed. In the 20^{th} century Wang had conjectured that no aperiodic sets existsed, where an *aperiodic* set was one which admitted only *non-periodic* tilings of the plane (i.e. no valid tiling of the plane is periodic). The first known set of aperiodic Wang tiles was discovered by R. Berger in 1966 [4]. Berger's original set had 20,426 prototiles. He later reduced this number to 104, and until recently, the smallest known aperiodic set had 16 prototiles [17].

Wang tiles are interesting theoretically because it is possible to find sets of Wang tiles that mimic the behavior of any Turing machine. They are interesting to us because sets of Wang tiles have been discovered with as few as 13 prototiles [9, 11]. Having fewer tiles is desirable because the geometry in each tile is very detailed and takes a lot of space to store.

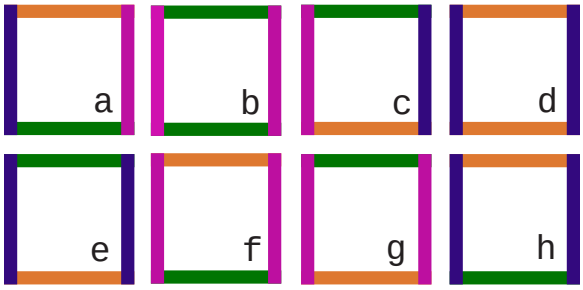
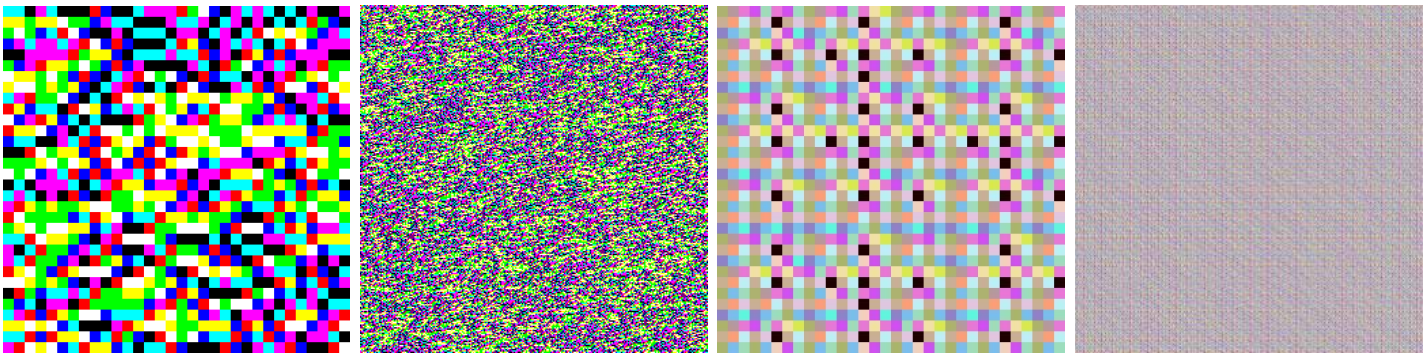


Figure 3: The set of 8 proposed prototiles

.

After our initial excitement at the idea of using Wang tiles, we found that although the tilings are aperiodic, at least the small Wang tile sets display a marked structure (see Figure 4(c)) which is exactly what we wanted to avoid. Rather than give up on this path we tried to create a small set of Wang tiles that could tile the plane simply and at least not appear periodic and not display any obvious structure. We found one such set of 8 tiles shown in Figure 3. A tiling is created with a very simply algorithm:

1. Choose a tile at random and place it in the lower left corner
2. For the bottom row, choose compatible tiles from left to right (i.e., the west edge must match the previous east edge). If more than one choice is possible, choose randomly amongst compatible tiles.
3. For each row above the bottom row
   (a) Choose the first tile to be compatible with the one below it (i.e., the south edge must match the north edge from below)



(a) (b) (c) (d)

Figure 4: Small scale (32x32) and large scale (256x256) tilings using (a and b) the 8 Wang tiles proposed in this paper and (c and d) Robinson's set of 16 aperiodic Wang tiles.

(b) Complete the row with tiles that match both the west and south edges, to tiles on the left and below.

Somewhat to our surprise this worked. A tiling of the plane with a random choice of color assigned to each prototile is shown in Figure 4(a). Larger scale tilings using our prototiles and Robinson's 16 are shown in Figure 4 (b) and (d) respectively. The large scale tiling of the Robinson tiles shows a marked plaid-like structure with strong horizontal and vertical features, while the large scale tiling of our tiles looks similar to white noise. Figure 2(d) shows a rendering of the sunflower scene modeled using this tile set.

This set of eight is clearly not strictly aperiodic as one could create a tiling of the plane using only one of the tiles that has the same color on its north and south edges, and on its east and west edges. However, our stochastic construction procedure prevents such a degenerate tiling from appearing in practice. We have generated valid tilings of ten thousand tiles on a side using our stochastic tiles.

3 Modeling

In our demonstrations, we model the terrain surface as a set of objects such as sunflowers, dandelions, and blades of grass in a random-close-packed arrangement. For visual variety, each type of terrain object has a number of versions. The sunflower scene is made up of 11 versions of the flower, while the grassy scene is made up of 15 versions of grass, 9 versions of the dandelion and 15 versions of the yellow flower. Each type of object has a radius that determines how densely they will be distributed.

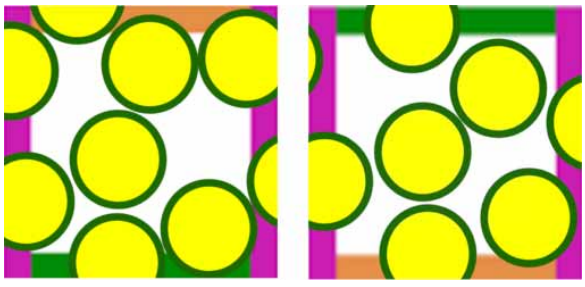


Figure 5: Poisson disk distributions for possible neighboring tiles.

We approximate this terrain model with an aperiodic arrangement of a small number of square tiles. For each of those tiles, we must construct a geometrical model made up of an arrangement of terrain objects that will be consistent with the tile's edge-color boundary condition. If one tile has a purple east edge, it could appear next to any one of several other tiles having a purple west edge (Figure 5). We would like the random-close-packed arrangement (i.e., a Poisson-disk distribution) to extend across the tile boundary no matter which of the purple-west-edge tiles happens to be adjacent. If the terrain object distributions in each tile do not mesh with neighboring tiles, the result is a highly visible, periodic disruption of the terrain along the grid lines between tiles.

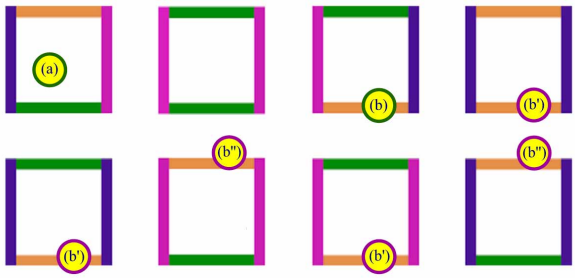


Figure 6: Placing objects in tiles.

To achieve consistency with the edge-color boundary conditions, we use a stochastic dart-throwing process to produce a set of points (terrain-object locations) in and around each tile. The dart-throwing process visits each tile in round-robin order and attempts to add one new object to that tile, until a sufficient density of objects have been placed in each. This round-robin processing insures that the tiles have almost identical object density. During each visit to a tile, up to 10,000 attempts are made to insert a new object at a uniformly distributed random location.

An attempt succeeds if the new object's radius does not overlap any other object's radius either within the tile or in potentially neighboring tiles. If this is true, it is added to the tile and we may move on to the next tile. When the new object's radius lies completely within the tile, Figure 6(a), the new object's radius need only be check against other points associated with that tile. If the new object's radius extends beyond an edge of the tile (e.g., an orange south edge), Figure 6(b'), then an attempt must be made to add this new object into

- all other tiles that have the same edge color for the same edge (e.g., all orange south edges), Figure 6(b') and
- and all potential neighbors across the edge (e.g., all orange north edges), Figure 6(b").

In the latter check, the new object's location is outside the tile as if it were in the neighboring tile (e.g., across the blue south edge). Note that tiles can now have points associated with them that will lie just outside their boundary; however only the portion of the corresponding terrain object that protrudes into the tile will be rendered. All of the above conditions must be true for the new object to be accepted.

This treatment of the edge-color boundary condition means that the distribution of terrain objects is identical near each colored edge. Since the tiles are aperiodic, these edges are randomly distributed and do not appear to create a periodic visual artifact in the final scene. However, there is one remaining problem. If a terrain object protrudes beyond a corner of a tile, it could overlap any or all other tiles in the tile set. That might force us to place a terrain object in the same location near the corner of all tiles, and that would produce a periodic artifact in the scene. To address this problem, we assign two radii to each terrain object. One radius tightly bounds the extent of the object's geometry, while a slightly larger radius is actually used to control the packing density, giving a slight buffer zone. We do not allow the tight radius of any terrain object to protrude across a corner of a tile, but we do allow the buffer to cross a corner. The result is that objects crowd in slightly to avoid creating a periodic void in the distribution at each tile corner.

We have used a simple terrain model based on a Poisson-disk distribution of plant species. Deussen et al. [6] present several more advanced models, based on plant population dynamics and terrain topography (elevation, slope, closeness to water, etc.). Modeling these phenomena presents an interesting, unsolved challenge to a real-time tile-based approach. Possible future work would experiment with using larger numbers of tiles and edge colors, and associtate a range of plant densities with each edge color. This may afford enough lattitude to adapt the tiling to local topographical conditions.

4 Representation

Once we have finished placing instances of the plant models in the tiles, we are faced with the task of creating a run-time representation of each tile. There are several characteristics we would want of such a representation. It should

- render as realistically as possible,
- look good at many different screen space resolutions,
- look good from many different angles, and
- be able to render at interactive rates.

Our solution is to build multiresolution view dependent Layered Depth Images (MRVDLDI), a collection of orthographic Layered Depth Images sampled at varying resolutions and from varying directions. Any particular tile, at a particular resolution and from a particular direction is used in the reconstruction only from an appropriate distance for its resolution, and from a small range of directions. Taken together, they can be relied on to fill the filed of view from any position, looking in any direction.

Layered Depth Images are a natural fit for all of the above requirements. LDIs are a sampled representation where the samples of geometry are chosen to optimize the reconstruction from a small range of viewpoints. Samples are chosen and individually rendered offline as a preprocess. Therefore the designer can choose an arbitrarily complex global illumination simulation to shade the samples. The choice of lighting simulation affects the quality of the final LDI, but does not affect the run-time performance of reconstruction.

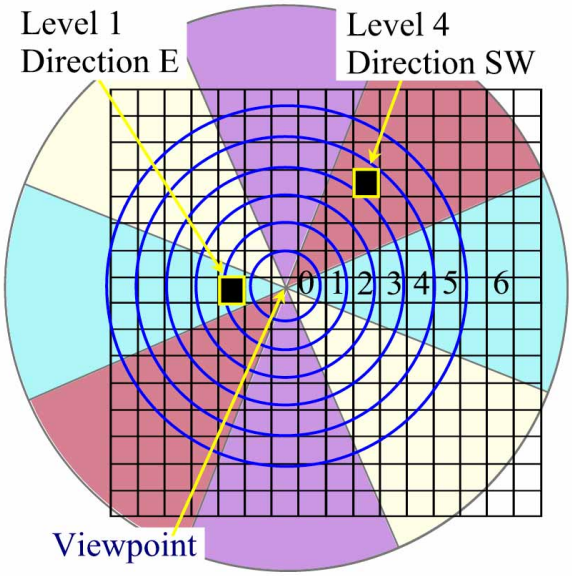
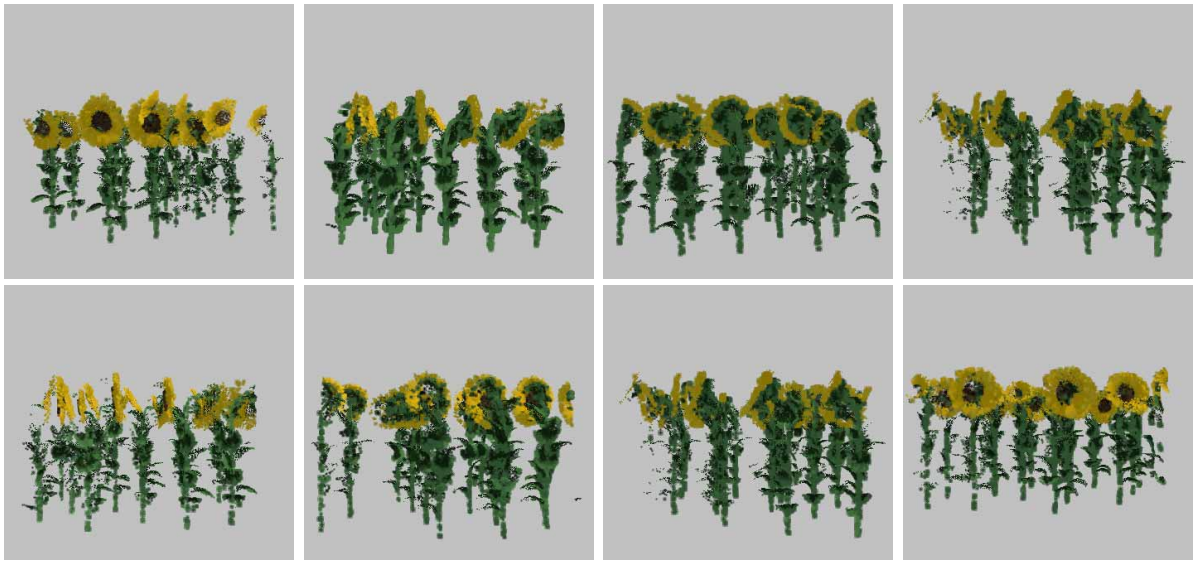


Figure 7: Levels of detail and choice of direction.

The image space parameterization of LDIs makes them a natural fit for creating level of detail hierarchies. We will build each LDI in the hierarchy individually rather than create a high resolution LDI and down sample it. By doing this we can optimize the sampling at each resolution. LDIs as described by Shade *et al.*[19], are purposely sampled to be best viewed from novel viewpoints near the center of projection of the LDI. In the absence of highly specular surfaces, the same point in nearby views will not change appreciably. Also, the sampling of the geometry of the scene is biased towards sending rays from the region near the LDI center of projection. So, there is an additional view dependence on the visibility complex captured in the LDI. The LDI data structure we present in this paper takes advantage of the visibility dependence, but only approximates view dependent lighting. We have worked solely with scenes that consist of diffuse objects. In the future work section we mention an extension to the LDI rendering algorithm that, when combined with view-dependent LDIs, would better reconstruct specular effects.

As already said, a multiresolution view dependent Layered Depth Image is formed from a collection of orthographic Layered Depth Images. Each LDI is sampled at varying resolutions and from varying directions. We chose to use an orthographic frustum because its rectilinear volume fits naturally into a square tile of terrain. The orthographic camera is configured to look top-down at a square plot of the solid texture we wish to tile across a terrain. Figure 11 shows top-down orthographic and off-to-the-side perspective views of the eight tiles used in the sunflower scene.

We will rely on the accuracy of each MRVDLDI only in the space of viewpoints from which we expect it to be viewed. Figure 7 shows a top-down view illustration of a virtual camera above a tiled terrain. Although not depicted in this diagram, the height of the viewer is chosen prior to LDI sampling to be about "head height". The same height is used by the run-time system to constrain the movement of viewer. We create a level of detail for each of the first seven tiles that radiate outward from the position of the viewer. Beyond this distance, there is very little parallax in the geometry of tiles and a directional texture (or a one layer LDI) can be used instead of a full 3D LDI. Figure 7 shows the concentric rings of level of detail chosen for a particular viewpoint at run-time. Figure 9 shows an illustration of the level of detail algortihm. On the left



(a) (b) (c) (d)

Figure 8: View dependent sampling. The preferred view direction is: (a) from the south (b) from the west (c) from the north (d) from the east.

is a tile far in the distance. If we freeze it's level of detail and dolly forwards, the low resolution of the LDI becomes apparent. On the right we show the proper level of detail for the same distance as the middle image. The results shown in this paper used LDIs of resolution 468, 232, 151, 111, 88, 73, and 62 pixels square.

In addition to a small range of distance, each MRVDLDI is expected to be viewed from only a small range of angles. We divide the circle of sample directions into evenly sized wedges that encircle the tile (shown in Figure 7). For each level of detail (distance to the viewer), eight directional LDIs are constructed. A directional LDI is simply an orthographic LDI with an associated preferred view direction. This view direction is used at run time in conjunction with the viewer's position to determine the appropriate directional LDI to render. Figure 8 shows the affect of directional sampling of the LDIs. In each column, the top picture is a view from the direction of the sampling rays. The bottom picture is a view rotated ninety degrees counter-clockwise about the center of the tile. So, the bottom row views are from the same angle as the images one column to the right in the top row. By comparing the top row images to the images one down and to the left, you can see the affects of the directional sampling. The side of the LDI opposite the side closest to the sampling rays is sampled more sparsely.

Our LDI sampling algorithm proceeds as follows. We position a camera at "head height" above the origin. For each tile, we determine the resolution of each level of detail by placing the bounding box of the tile at the appropriate distance from the viewer. The camera is tilted to look at the center of the top of the bounding box (approximately the gaze we expect the viewer to have at run time). The screen space projection of the top of the bounding box is then computed, telling us the resolution of the LDI required to ensure a splat size of one. In other words, the maximum dimension of the screen bounding bow defines LDI tile's resolution. Once the resolutions are determined, all of the LDIs for a tile are computed by: for every level of detail and for every direction, rotate the tile about its center the appropriate amount, translate the tile to the appropriate distance from the origin, and sample the LDI using the stochastic ray casting scheme described by Shade et al.[19].

The result of this process is a series of progressively coarser levels of detail which consist of eight directionally biased LDIs. An obvious question to ask is: why not throw all of the samples for a single level of detail into a single LDI and cull back facing pixels at run-time? The reason is that the speed of LDI warping is very sensitive to the memory coherence of the depth pixels. Since depth pixels are parameterized along the rays of the image, depth pixels that are front facing (from any particular view) are going to be intermingled with many back facing pixels. Skipping over the back facing pixels at run-time would eliminate any positive effects of cache coherence gained by the compact nature of the LDI data structure (adjacent depth pixels along a ray are stored next to each other in memory). By factoring each level of detail into eight directional LDIs, we are tuning each component LDI to render quickly from its preferred direction.

To further reduce the size of the directional LDIs (which enables them to render more quickly in addition to saving space), we take advantage of the occlusion characteristics inherent in the geometry of the tiles. When creating the LDIs for a tile, we place the tile "in context". For every tile, we compute a 3x3 tiling, with the target tile in the center. The geometry for all of these tiles are then used in the procedure described above: any sampling ray that hits geometry outside of the volume of the target tile is discarded. By surrounding the target tile with a plausible tiling, we are approximating the occlusions that occur at run time.

(a) (b) (c)

Figure 9: Level of detail. A tile rendered at a low level of detail: (a) rendered at the correct distance (b) the same LOD, close up (c) the proper close up LOD

5 Real-time Rendering

We have implemented an interactive renderer that combines a software based Layered Depth Image warper with a simple OpenGL based polygon renderer. The system is fast enough to allow the user to move around in real-time. After a short initialization step, rendering proceeds in three stages:

- LDI rendering producing an image with alpha plus a z-buffer,
- polygon rendering of the terrain or other standard graphics objects with OpenGL, and finally
- z-compositing the LDI image over the OpenGL image to account for transparency in the LDI rendering.

5.1 Initialization

The input to the interactive renderer is: a set of prototiles, a set of multiresolution view-dependent layered depth images (one set per prototile), a terrain height field, and the initial position of the viewer. The first thing the system does is compute a tiling that covers the entire height field. Not every tile in the tiling has to be instantiated. For instance, in the examples we show, only tiles that lie on parts of the terrain below some threshold height are instantiated. This is done to prevent the system from trying to put tiles of flowers on tops of mountains. One could, instead, decide to place tiles based on the gradient of the height field or use an image mapped over the terrain that is painted by hand to define where tiles could be instantiated [6].

In preparation for rendering, our system finds the object space bounding boxes of all of the tiles in a scene. The footprint and height of the tiles are determined in the modeling phase. The world space placement of a tile's bounding box is determined by the mapping of the tiling onto the height field.

5.2 LDI Rendering

For each frame, the first stage of rendering uses a Layered Depth Image warper to create an image of all of the visible tiles. To render the visible set of tiles, we step through the tiling in a back-to-front order (as determined by the viewer's position and orientation). Each tile's object space bounding box is tested for inclusion in the view frustum. If this test succeeds, the level of detail for the tile is computed by finding the distance between the tile and the viewer. Lastly, the most appropriate directional LDI at this level of detail is chosen. The dot product of the vector from the viewer's position to the tile and the preferred direction vector of each directional LDI's is computed. The directional LDI whose vector has the lowest dot product is the one chosen. This LDI is then transformed so that it coincides with the position of the tile and warped into the LDI frame buffer.

Since this image is going to be combined with a hardware accelerated rendering of the terrain we also compute a z-buffer as a side effect of the Layered Depth Image warping. This is a straightforward extension to the algorithm in [19]. We render using a back-to-front order of the tiles, and within each tile, we use McMillan's occlusion compatible warp ordering [12]. The projective z of every pixel already computed for the splatting calculation is written into a software z-buffer at the end of the warping function. Since we are not using the z-buffer to determine visibility, it may get written multiple times. In fact, it will get overwritten exactly the same number of times the color buffer does, with the final write containing the closest z value.

5.3 Terrain Rendering and Z-compositing

Once all of the tiles have been warped, the terrain is then rendererd using OpenGL. The only acceleration technique we use on the terrain is to render it in triangle strip order. The z-buffer produced by the LDI code is then written into the z-buffer produced by the terrain rendering. At every pixel where the an LDI z value is closer than the a terrain z value, a bit it set in the stencil planes. We then alpha composite the color buffer from the LDI rendering using the over operator, configuring OpenGL to only modify the color buffer at the pixels where the stencil buffer has been set. This properly z-composites [7] the depth image from the LDI rendering onto the depth image created by the OpenGL rendering.

Unfortunately, reading or writing the z-buffer in OpenGL is very slow. On our hardware, a 3DLabs Oxygen GMX 2000, this can be done at a maximum rate of 5 to 9 frames per second. This is the primary bottleneck in the system. LDI rendering alone runs at an average of 3 to 6 frames per second, and the terrain can be rendered at 30 frames per second. We don't see any reason why this should be slow other than it is not a typically optimized path in an OpenGL driver. Hopefully future hardware will overcome this.

5.4 Shear Warping LDIs

In order to mold the LDIs to the terrain, we add an affine shear warp to the standard LDI rendering algorithm. To facilitate this, we render the LDI in two triangular sections. The shear warp is straightforward to compute: we define a frame with the world up mapped to the up vector, and each of two sides of a terrain triangle mapped to the other two vectors. The frame defines the matrix used to do the shearing. Lastly, we add a separating plane to each tile that runs along the diagonal of the LDI. At runtime we use this plane and the viewer's position to determine a back-to-front drawing order of the two halves of the LDI. Shearing is just an approximation to the true deformation that would exactly mimic placing the objects in the tiles on the terrain. If the terrain is very steep or has a sharp feature, the LDIs can gent bent in unnatural ways. As our results show, for a genty rolling terrain, the shear warp provides an adequate approximation.

6 Results

We demonstrate our system using two sets of tiles, a field of sunflowers and a field grass. Each of the tile sets is mapped onto a synthetic terrain of gently rolling hills. Figures 11 and 12 show top-down orthographic and off-to-the side perspective views of each of the tiles. The orthographic views are render with shadows turned off in order to make it clear where every object in the tile is being placed. In addition, these view were rendered with the tile "in context" in order to show the edge constraints. The off-to-the side views were rendered with only the objects assigned to each tile. Objects from adjacent tiles that may overlap the edges are not shown.

Figure 10 shows screen shots from our interactive renderer. The top 6 pictures show the sunflower tiles being mapped over the terrain, while the bottom six show the grassy tiles being mapped over the terrain. These are very complex data sets. Each sunflower tile holds on average 20 sunflowers, and each sunflower is comprised of 35,000 triangles. At any point in time, there are approximately 7,000 flowers in the view frustum. This means that our system is reconstructing a view of a database of 245 million triangles. Our viewer can render this scene at 1.5 to 3 frames per second. If we don't render the terrain (and thus avoid the OpenGL bottleneck) our system can render just LDI's at a rate between 2 and 9 frames a second. The system used to do the timings was a Dell

Precision 610 workstation with a single 550MHz Pentium III, 512 MB of memory, and a 3DLabs Oxygen GMX 2000 graphics card.

The multiresolution view dependent LDIs for this scene require 195 MB of storage. The highest resolution level of detail alone is responsible for 140 MB. Typically, only the closest 3 or 4 tiles are rendered at this level of detail. This leads one to the logical conclusion that a geometry based solution for the closest tiles deserves attention. In order to render the 4 closest sunflower tiles with geometry at 10 frames per second, we need a graphics card that deliver a sustained polygon rendering rate of 28 million triangles per second. Graphics card with this level of performance should be available in the next two years.

The tiles in the grassy scene are comprised of grass, dandelions and yellow flowers. Each tile in this scene has approximately 62,000 triangles, and the view frustum intersects a portion of the scene with 25 million triangles. The multiresolution view dependent LDIs for this scene occupy 278 MB of memory. We get run time performance of only 1 to 2 frames per second for this scene.

Lastly, Figure 1 shows a rendering using one degree digital elevation data distributed by the USGS. While this data set does not demonstrate the shear warping of the tiles, it does answer the question: What would Yosemite valley look like if it was covered in sunflowers?

7 Discussion

Several aspects of our work merit further discussion. The first is modeling. Figures 12 shows top-down views of the grassy tiles. An obvious artifact in these tiles is that that corners are unnatrually sparse. This is inherent in two dimensional tilings. Even with edge contraints, the tile that is across a corner is unconstrained. Therefore, no object can cross two borders of a tile. Neyret recognized this same problem [16] and solved the problem by making the corner of every tile the same color. We could do the same thing here by manually placing objects in the corner before stochastic object placement. In addition our stochastic tiling scheme has the undesirable property that some local clumping can occur. In Figure 4(a) you can see areas where a bunch of tiles made from the same prototile clump together in a bunch. This can produce unnatural looking results. Designing a new tile set may alleviate this, or we can extended the tiling algorithm to never put two tiles of the same type next to each other.

The LDI creation phase of our system is computationally very expensive. For each directional LDI we are casting 2.5 million rays. This amounts to one billion eye rays to sample all of the tiles. Creating the sunflower data set takes a single 500 MHz Intel processor 64 hours. The grassy scene takes 288 hours.

Our run-time system does not render every tile that intersects the view frustum. It only renders those tiles within a radius of 40 tiles from the viewer. There are two reasons for this: our system could not render the scene in real-time if we went all the way to the horizon; and, there is no reason to do so. The parallax in tiles that far away is minial, and would be best rendered as view dependent texture maps. We plan to implement this extension to our system to enable us to render a seamless terrain texture.

Lastly, this work indicates that sparse volume rendering representations are a viable alternative to geometry. Moore's law is on our side. As processors get faster, this technique will get correspondingly faster. Hardware support for a sparse volume rendering primitive would allow scenes like the ones we show here to be rendered at high frame rates. Even though the data sets we've created are quite large, only a small subset of the view dependent LDIs are needed for any frame. Furthmore, the set of LDIs needed changes smoothly and predictably as the viewer moves through the scene. It is conceivable that streaming the LDIs from system memory to the graphics card as needed would be possible.

8 Future work

There are many ways to extend this work:

- *Directional textures.* Foremost, we plan on implementing directional textures for extending the tiling as far as the eye can see. A seamless texture that extends to the horizon can be made by rendering eight directional texture maps in a manner simlilar to the directional LDI creation. Since these texture maps will only be used for very distant parts of the terrain, they can be very low resolution, say 32x32.

- *Blending LDIs.* Transitions between levels of detail and between different directional LDIs can result in popping. A way to combat this would be to blend between the two LDIs as the transition is made. In addition, blending would enable us to handle directional lighting, albeit at a coarse resolution.

- *Pipelined rendering.* Pipelining could possibly be used to help alleviate the bottleneck from slow OpenGL depth buffer blitting. However, since our demo runs at a fairly low frame rate, lag from user input would be significant.

- *Fairing the edges of the tiling.* The transitions from textured to non-textured parts of the terrain (like the tops of hills) is rather abrupt. One solution is to add special tiles whose textures don't cover the entire tile. This, of course, would increase the size of the tile data set.

- *Adding variations on tiles.* Another modeling option is to add more tiles who have the same edge constraints as one of the eight prototiles, but uses a different texture on the interior. This would allow us to put other objects in the tiles, such as trees.

- *Combining tiling with other walkthrough techniques.* This paper presents an effective technique for using Layered Depth Images as the "middle ground" primitive in combination with other techniques such as image caching and geometric-based multiresolution models.

9 Conclusions

We have presented a technique that enables real-time rendering of highly complex three dimensional scenes. By combining a well known result from the field of two dimensional tiling with a recent advance in image-based rendering, we have provided a solution to a long sought after problem: adding realistic three dimensional texture to terrains. This technique of computing a two dimensional tiling of three dimensional tiles is very powerful. There are many ways in which the solution to a 3D problem can be approximated using this 2.5D technique. Examples include:

- *Automatic creation of dungeons or mazes.* An obvious application to games is using tiling to automatically create dungeon-like mazes. A tile would consist of several levels in the dungeon. Edge constraints would require that passages or rooms along the edges match. There is great lee way in the granularity of the tile in this situation. If the tiles are made large enough, there could be variations on the interiors of the tiles. In other words, having several versions of every tile, all with the same edges but with unique interiors.

- *Approximation of three dimensional simulations.* Any simulation that is expensive to compute could be approximated on a large scale by tiling smaller scale simulations. An example is computing a fluid flow simulation. The simulation could be solved locally inside each tile, taking care every few time steps to make sure the boundaries between the tiles agree. This would certainly not produce a correct fluid flow simulation, but it may produce a plausible one with modest compute resources.

References

[1] Daniel Aliaga, Jonathan Cohen, Andrew Wilson, Hansong Zhang, Carl Erikson, Kenneth Hoff, T. Hudson, Wolfgang Strzlinger, Eric Baker, Rui Bastos, Mary Whitton, Frederick Brooks, and Dinesh Manocha. Mmr: An interactive massive model rendering system using geometric and image-based acceleration. *1999 ACM Symposium on Interactive 3D Graphics*, pages 199–206, April 1999. ISBN 1-58113-082-1.

[2] Daniel G. Aliaga and Anselmo Lastra. Automatic image placement to provide a guaranteed frame rate. *Proceedings of SIGGRAPH 99*, pages 307–316, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.

[3] Animatek. *World Builder Web Site*. http://www.animatek.com, 1999.

[4] R. Berger. The undecidablility of the domino problem. In *Memoirs Amer. Math. Soc.*, page 72, 1966.

[5] Meta Creations. *Bryce Web Site*. http://www.metacreations.com/products/bryce4, 1999.

[6] Oliver Deussen, Patrick Hanrahan, Bernd Lintermann, Radomír Mech, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. *Proceedings of SIGGRAPH 98*, pages 275–286, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[7] Tom Duff. Compositing 3-d rendered images. *Computer Graphics (Proceedings of SIGGRAPH 85)*, 19(3):41–44, July 1985. Held in San Francisco, California.

[8] Branko Grunbaum and G. C. Shephard. *Tilings and Patterns*. W. H. Freeman and Company, 1987.

[9] Karel Culik II. An aperiodic set of 13 wang tiles. *Discrete Mathematics*, 160:245–251, 1996.

[10] James T. Kajiya and Timothy L. Kay. Rendering fur with three dimensional textures. *Computer Graphics (Proceedings of SIGGRAPH 89)*, 23(3):271–280, July 1989. Held in Boston, Massachusetts.

[11] Jarkko Kari. An small aperiodic set of wang tiles. *Discrete Mathematics*, 160:259–264, 1996.

[12] Leonard McMillan. Computing visibility without depth. Technical Report 95-047, University of North Carolina, 1995.

[13] Radomír Mech and Przemyslaw Prusinkiewicz. Visual models of plants interacting with their environment. *Proceedings of SIGGRAPH 96*, pages 397–410, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[14] Alexandre Meyer and Fabrice Neyret. Interactive volumetric textures. *Eurographics Rendering Workshop 1998*, pages 157–168, June 1998. ISBN 3-211-83213-0. Held in Vienna, Austria.

[15] Fabrice Neyret. A general and multiscale model for volumetric textures. *Graphics Interface '95*, pages 83–91, May 1995. ISBN 0-9695338-4-5.

[16] Fabrice Neyret and Marie-Paule Cani. Pattern-based texturing revisited. *Proceedings of SIGGRAPH 99*, pages 235–242, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.

[17] R. M. Robinson. Undecidable tiling problems in the hyperbolic plane. In *Inventiones Math.*, pages 259–264, 1978.

[18] Gernot Schaufler and Wolfgang Stürzlinger. A three dimensional image cache for virtual reality. In *Proceedings of Eurographics '96*, 1996.

[19] Jonathan Shade, Steven J. Gortler, Li wei He, and Richard Szeliski. Layered depth images. *Proceedings of SIGGRAPH 98*, pages 231–242, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[20] Jonathan Shade, Dani Lischinski, David Salesin, Tony DeRose, and John Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. *Proceedings of SIGGRAPH 96*, pages 75–82, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[21] Jos Stam. Aperiodic texture mapping. Technical Report R046, European Research Consortium for Informatics and Mathematics (ERCIM), 1997.

[22] Verant. *Everquest Web Site*. http://www.verant.com, 1999.

[23] Hao Wang. Proving theorems by pattern recognition. *Bell system Tech. J.*, 40:1–42, 1961.

[24] Hao Wang. Games, logic and computers. *Scientific American*, pages 98–106, November 1965.

[25] Jason Weber and Joseph Penn. Creation and rendering of realistic trees. *Proceedings of SIGGRAPH 95*, pages 119–128, August 1995. ISBN 0-201-84776-0. Held in Los Angeles, California.

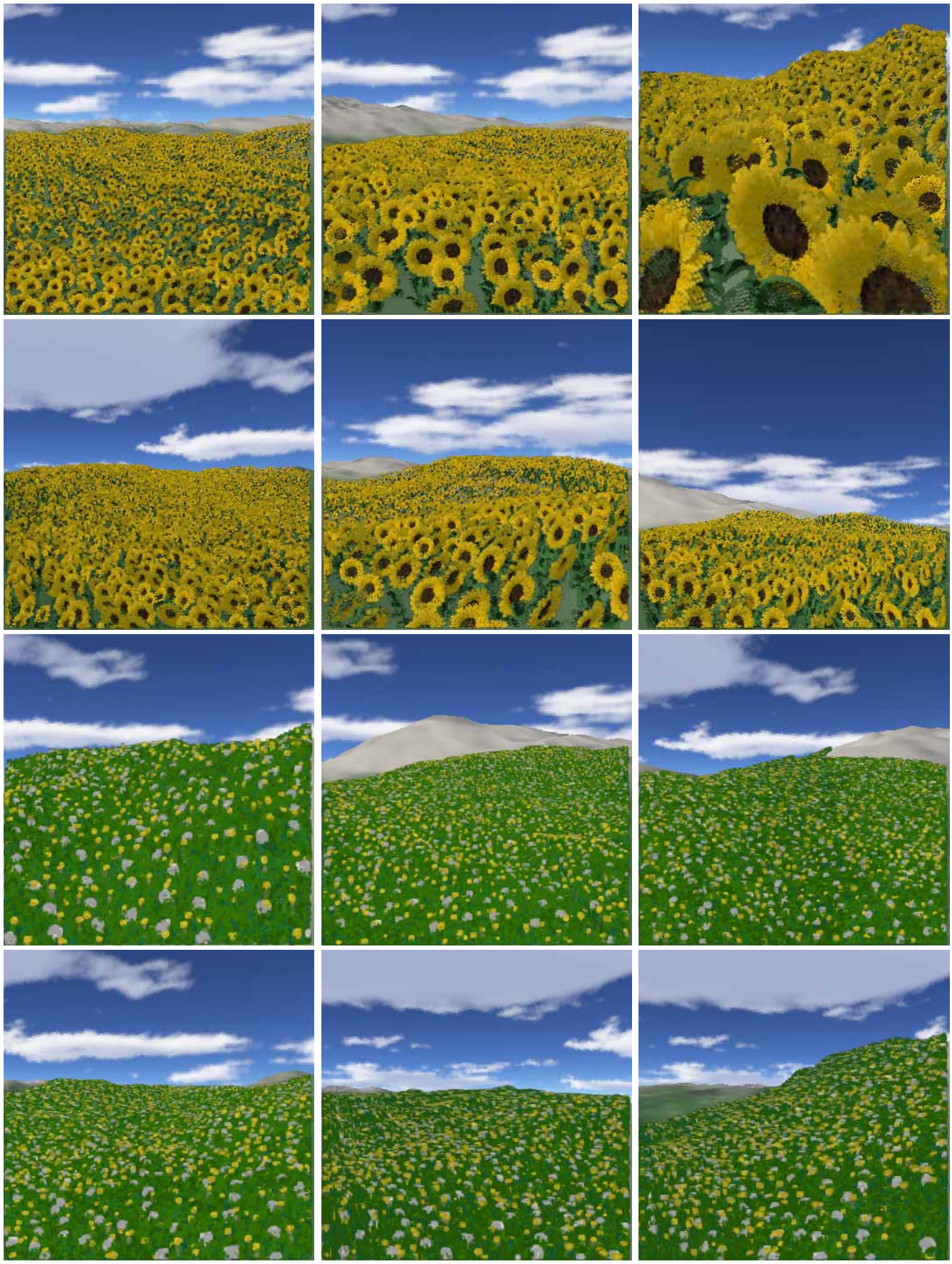


Figure 10: Screenshots of our real-time renderer in action.

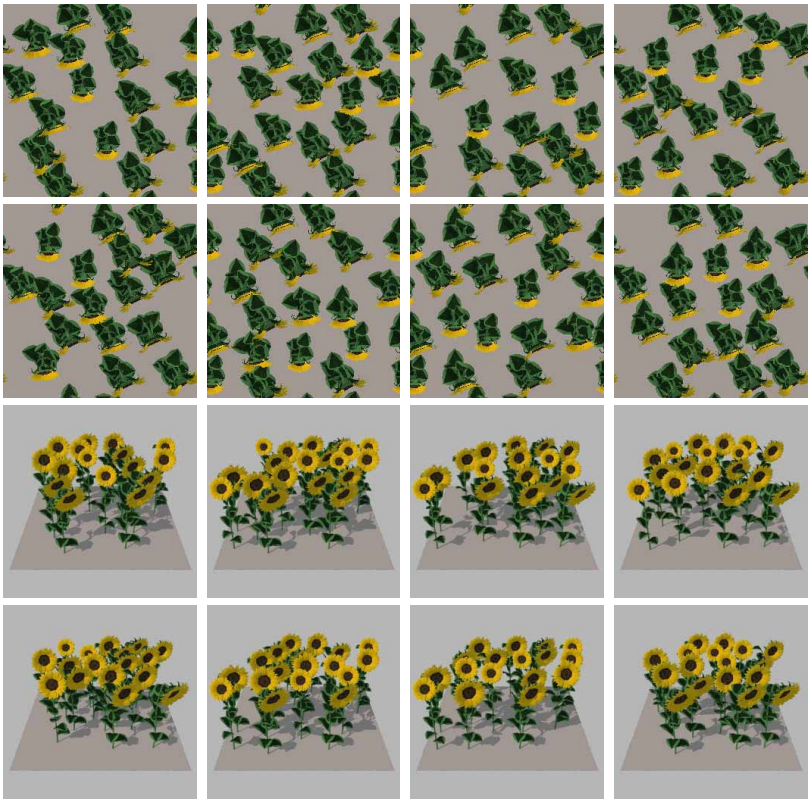


Figure 11: The 8 tiles used for the sunflower terrain in top-down and perspective views.

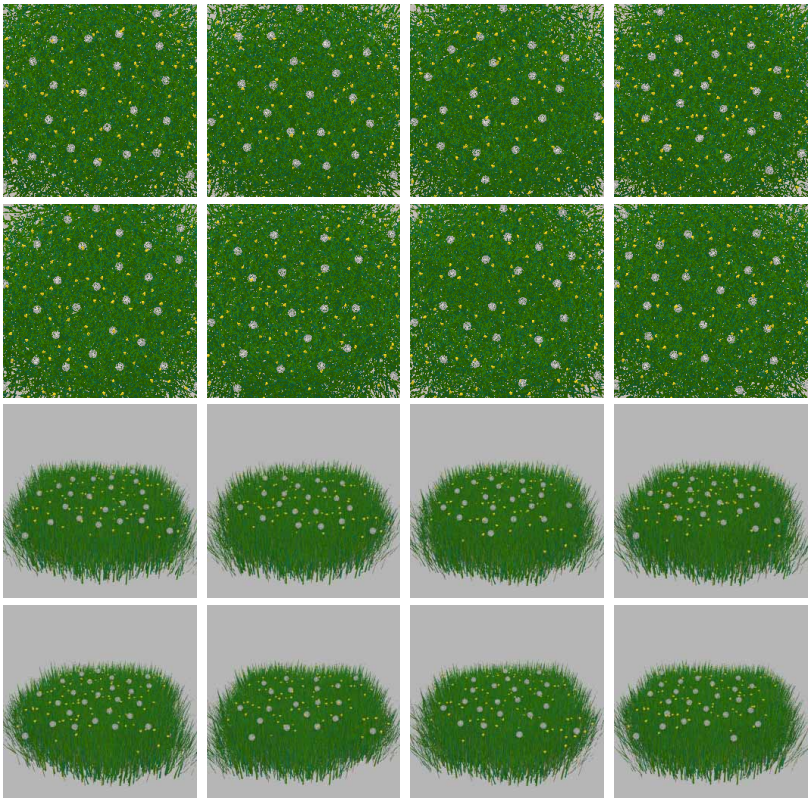


Figure 12: The 8 tiles used for the grassy terrain in top-down and perspective views.