EUROGRAPHICS 2012 / P. Cignoni, T. Ertl
(Guest Editors)

Volume 31 (2012), Number 2

Just-in-Time Texture Synthesis

paperID:1022

Figure 1: *(Left) umbrella diversification: the columns show, from left to right, the 4 base umbrellas, color modification, 2-D shape modification, and both color and 2-D shape modification. (Middle) texture synthesized with our method by diversifying the 4 base umbrellas. (Right) texture with same arrangement but only using the 4 base umbrellas, for comparison.*

Abstract

Texture bombing is a texture synthesis approach that saves memory by stopping short of assembling the output texture from the arragnement of input texture pataches; instead, the arrangement is used directly at run time to texture surfaces. However, several problems remain in need of better solutions. One problem is improving texture diversification. A second problem is that mipmapping cannot be used since texel data is not stored explicitly. The lack of an appropriate level-of-detail (LoD) scheme results in severe minification artifacts. We present a just-in-time texturing method that addresses these two problems. Texture diversification is achieved by modeling a texture patch as an umbrella, a versatile hybrid 3-D geometry and texture structure with parameterized appearance. The LoD is adapted with a hierarchical algorithm that acts directly on the arrangement map. Results show that our method can model and render the diversity present in nature with only small texture memory requirements.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism-Texture—Color, shading, shadowing, and texture

1. Introduction

Texture mapping is a uniquely powerful method for enhancing surface appearance in interactive computer graphics. Texture synthesis research efforts have produced techniques that construct high resolution textures based on input texture patches and patterns. Unfortunately, the high resolution synthesized texture requires large amounts of texture memory. Texture bombing addresses this challenge by stopping short of assembling the high-resolution texture from the arrangement of patches; instead, the arrangement is used directly at run time to texture surfaces. The texture is synthesized just in time and is never stored explicitly, which brings considerable texture memory savings.

However, several problems related to texture bombing remain in need of better solutions. One problem is improving texture diversification. Given a small number of input patches, a plausible large texture can only be synthesized if the appearance of the input patches is modulated sufficiently to reflect the diversity present in nature. A second problem is that mipmapping cannot be used since texel data is not stored explicitly. The lack of an appropriate level-of-detail (LoD) scheme results in severe minification artifacts.

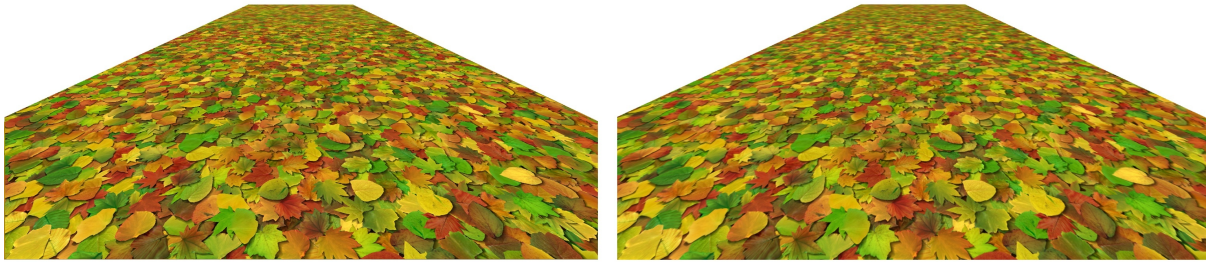

Figure 2: *Just-in-time texture synthesis (left, 9MB) and conventional texture of equivalent resolution (right, 190MB).*

Figure 3: *3-D shape diversification (top) and comparison between texture with and without 3-D detail.*

In this paper we present a just-in-time texture synthesis method that addresses these two problems. Texture diversification is achieved by modeling a texture patch as an *umbrella*, a versatile hybrid 3-D geometry and texture structure with parameterized appearance. The input patch umbrellas are modified and arranged to synthesize a large, high-resolution, and diverse texture. The input patch is modified substantially by interpolation to new colors and 2-D and 3-D shapes. In Figure 1 the 4 base umbrellas are sufficient to create hundreds of unique modified umbrellas (left), so the synthesized texture (middle) does not suffer from repetitiveness, which would be readily noticeable if the texture were synthesized only from the 4 base umbrellas (right). Modifying umbrellas to assume convex and concave 3-D shapes allows adding 3-D detail to the synthesized texture (Figure 3).

For the second problem we propose a hierarchical LoD algorithm for just-in-time texturing that acts directly on the arrangement map. Lower LoD arrangement maps are computed offline by merging umbrellas and are used at run-time to avoid minification artifacts. The result is quality similar to that of conventional mipmapping at a fraction of the storage cost (Figure 2). Please also see the accompanying video.

2. Related Work

A variety of texture synthesis methods have been developed [WLKT09]. Methods can be classified according to the periodicity of the data of the generated texture, which can be regular, such as a brick wall, irregular, such as fallen leafs on the ground, or purely stochastic, such as a rough surface. Regular textures have been modeled procedurally [LP00]. Other methods separate the sample texture into a regular and an irregular component, e.g. by using fractional Fourier analysis [NMMK05], which are then modeled independently, diversified and combined during texture synthesis [LTcL05]. We target the synthesis of irregular textures.

Texture synthesis methods can also be classified as procedural or sample-based methods. Procedural methods use the input texture to derive a complex model that allows synthesizing new textures of the same type as the input provided, e.g. based on Markov Random Fields [Pag04]. Sample-based methods, like ours, assemble the texture from modified versions of the input patches. A sample-based texture synthesis method needs to address three tasks.

The first task is to extract the texture patch from input images. Extraction is usually done with the help of image processing techniques [DMLG02, ZZV*03, WY04, LH06], but can also proceed through random selection of a rectangular window [LLX*01, KSE*03]. The second task is to arrange the extracted texture patches in the output texture domain. Some textures require an overlapping arrangement, which can be achieved by random placement of patches while controlling patch density [DMLG02, HQXT05]. Other textures require a seamless tiling of patches, achieved using graph cut techniques [KSE*03, EF01], patch stitching [DLC05], wang tiles [CSHD03], or sparse linear system optimization [PFH00]. The arrangement is either learned from an example [IMIM08, MWT11], random [KCoDL06, TW08], or defined with the help of user input [LN03]. The third task is to diversify a small number of input patches to convey the diversity present in nature. Diversification methods rely on many-knot spline interpolation [HQXT05], on regular lattice combined with deformation fields [LLH04], on texture meshes inspired from image meshes [DZ06], or on multi-scale descriptors which allow for appearance-space jitter that retains the structure on the input texture patches [RHDG10].

Texture bombing–the idea of saving memory by reusing a few texture patches placed at random locations–was pioneered over thirty years ago [SA79]. The advent of programmable graphics hardware brought renewed interest in

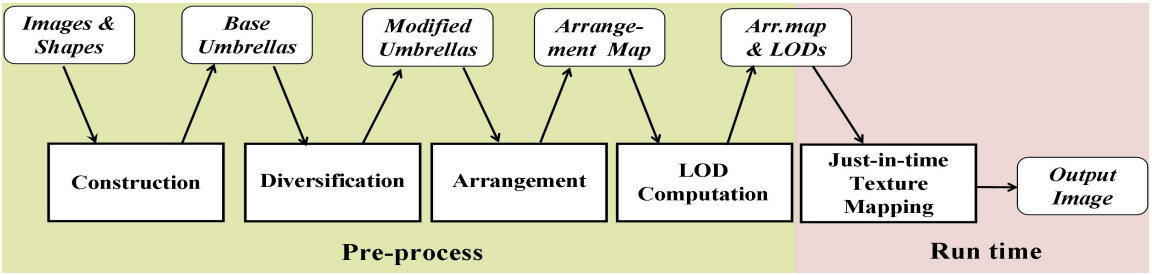

Figure 4: *Overview of the just-in-time texture synthesis pipeline.*

the approach [Gla04]. Texture sprites [LHN05] are image patches projected onto geometry at run-time, which bypasses the need of a global planar parameterization. The texture bombing approach can be used in conjunction with any patch arrangement method, including wang tiles [CSHD03, Wei04, LD05], lapped textures [PFH00], and user input based [LN03]. Since texture bombing renders directly from patches, mipmapping [Wil83] cannot be used across patches as required for high levels of minification.

The goal of our work is the development of a texture synthesis method for real time rendering that achieves good diversity of the output texture without requiring considerable texture memory resources. We rely on prior art solutions for patch extraction and arrangement, while we focus on achieving powerful patch diversification through color, 2-D, and 3-D shape diversification and we introduce and LoD algorithm for texture bombing that supports arbitrary minification.

Our method saves texture memory by taking the texture bombing approach. Texture memory can also be saved by compression at texel level. The main approaches are based on block partitioning [KE02, SR06], on vector quantization [BAC96, TF08], and on wavelets [BIP00, DCH05, STC09]. All of these techniques allow looking up the compressed texture directly (e.g. [DCH05]). Compared to texture compression, our method achieves compact storage while avoiding compression artifacts: powerful diversification allows creating a large texture from only a small number of input patches, which are stored uncompressed.

Several procedural geometric modeling methods target foliage specifically. The methods rely on L-Systems [RSL*02, PTMG08], on probabilistic [DGAG06] algorithms, on diversification of low-count polygonal models [MGGA10], and/or on particle systems [RCS04] to simulate ecosystems and autumn scenery. Compared to these methods, our technique achieves diversification based on examples and not based on rules, and our technique generates a texture defined compactly in a 2-D domain as opposed to a 3-D geometric model which needs to be processed in its expanded form.

Finally, our method captures 3-D surface detail. Previous techniques include bump mapping [Bli78], horizon mapping [Max88, SC00, HDKS00], displacement mapping [Coo84, KS01], view dependent displacement mapping [WWT*03], parallax mapping [KTI*01], and relief texture mapping [POC05]. Compared to these techniques, our method trades 3-D modeling fidelity for rendering efficiency by mapping an umbrella patch to an ellipsoid. The ellipsoid can be rendered efficiently on the GPU [Gum03]. Moreover, our method only renders 3-D detail where needed and transitions smoothly from 3-D to 2-D.

3. Just-in-time texture synthesis overview

The texture is synthesized offline in four major steps (Figure 4). First, a small number of base umbrellas (e.g. 4 in Figure 1) are constructed from input images and shapes containing the desired texture elements. Second, the base umbrellas are diversified to hundreds of unique modified umbrellas by varying color, 2-D shape, and 3-D shape parameters. Third, the modified umbrellas are arranged in the 2-D texture domain. Umbrella construction, diversification, and arrangment are described in Section 4. Fourth, the umbrellas and the arrangement map are fed into an algorithm that computes the LoDs needed to accommodate any minification level (Section 5). The umbrellas, the arrangement map, and the LODs are then used at run-time to texture surfaces as needed for the current output image (Section 6).

4. Umbrella Texture Patches

4.1. Construction

We define an umbrella as a texture-mapped 2-D geometric primitive with a central vertex C and peripheral vertices V_i (Figure 5, left). The umbrella need not be convex, but all segments V_iC have to be inside the umbrella. The umbrella is a flexible light-weight representation that captures many texture elements present in nature with high fidelity.

The texture of the base umbrella is derived from input images that contain the desired texture elements. We construct base umbrellas with an interactive editor. The user first lays down the polyline defining the contour of the umbrella and then selects the center. For texture elements where features converge, the center is chosen at the convergence point for improved diversification results as discussed below. For example in the case of a leaf (Figure 5, left) the center is chosen at the convergence of the leaf veins. For the berry example

Figure 5: *Base umbrellas (left) and 3-D shape modif. (right).*

Figure 6: *Umbrella color modification.*

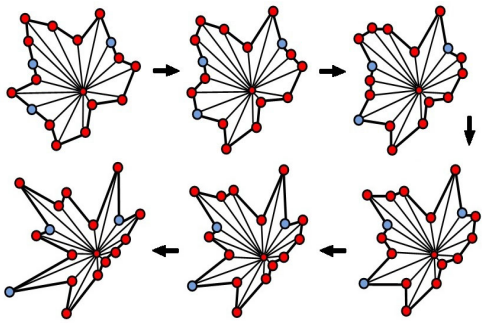

Figure 7: *2-D shape diversification by morphing.*

the center is simply chosen as the centroid of the peripheral vertices. The interactive editor allows creating a base umbrella in seconds. Only a few umbrellas are needed (e.g. 4-10), which are then diversified automatically.

4.2. Diversification

In order to modify the color of a base umbrella, its vertices are assigned colors that are used to modulate the texture of the umbrella. The color c_p at a point P inside the umbrella is computed as follows:

$$\begin{aligned} c_p &= c_t + f c_v \\ c_v &= \sum d_i c_i / \sum d_i \end{aligned} \tag{1}$$

where c_t is the color looked up in the base umbrella texture and c_v is a weighted average of the vertex colors c_i. A weight d_i is defined as an inverse of the distance between vertex i and P. The coefficient f controls how much the original texture colors are modified. c_p is clamped to [0,1].

We set the vertex colors using additional reference images of similar texture elements. In Figure 6, an image of a leaf with different colors (left) was used to set the colors of the peripheral vertices of the base umbrella (middle) which yields a realistic leaf with significantly different colors, at only a small additional storage cost.

The 2-D shape of a base umbrella is modified by moving peripheral vertices. A vertex can move to any new location as long as this does not create a fold. In order to allow for complex shape modifications, some base umbrella edges might be split into multiple segments. In Figure 7 the base umbrella has collinear peripheral vertices (blue) which can move progressively to create significantly different leaf shapes. The shape morph is specified with a second position for each of the peripheral vertices. These positions can be designed by the user, or can be derived from the shapes of other leafs. The morph produces a large number of plausible 2-D leaf shapes without a significant storage cost increase.

We support modeling of 3-D surface detail by mapping umbrellas to ellipsoids (Figure 5, right), which provide a good tradeoff between modeling power and rendering cost. 3-D shape is diversified using curvature magnitude and direction (i.e. convex or concave).

4.3. Arrangement

The texture is synthesized by arranging modified umbrellas in the 2-D texture domain. Depending on the texture a tiling or an overlapping arrangement is needed. The umbrella shape is sufficiently flexible to be compatible with previously developed arrangement methods. We exemplify our just-in-time texturing method using an overlapping arrangement defined using a regular grid. Modified umbrellas are assigned to grid cells. A grid cell is assigned all umbrellas that intersect it (Figure **??**). The umbrellas are stored in back to front order. For the examples shown in this paper the base umbrella color and shape diversification parameter values, as well as the location, rotation, and scale of the modified umbrellas are chosen randomly.

5. Level of detail pre-processing

We adapt the level of detail in two ways. First, expensive 3-D detail should only be rendered where it matters, i.e. close to the eye. This requires switching gradually from 3-D detail to a flat surface. Second, when umbrellas have a small image footprint, mipmapping individual umbrella textures is not sufficient to avoid minification artifacts, and umbrellas have to be merged. Whereas traditional texture synthesis methods actually compute a large texture and individual patches are merged implicitly through mipmapping, just-in-time texture synthesis requires merging umbrellas explicitly to compute coarser LoDs of the arrangement grid. Figure 8 illustrates how, at the highest level of detail, the synthesized texture is rendered with full 3-D detail, then the height of the 3-D detail is tapered off gradually, and then coarser and coarser LoDs of the arrangement grid are used. Whereas tapering off 3-D detail can be done at run-time, the coarser LoDs of the arrangement grid have to be pre-computed offline, akin to pre-computing the mipmap levels of a conventional texture.

We pre-compute coarser levels of detail of the arrangement grid hierarchically from the bottom up. Consider a grid

with at most k modified umbrellas per grid cell. The next coarser level is computed by merging 4 neighboring cells into 1 cell with k umbrellas using Algorithm 1.

Algorithm 1 *Arrangement map LoD computation.*

```
Group up to 4k umbrellas into k clusters
for each cluster C_i do
    Compute center o_i of C_i
    Compute convex hull h_i of C_i
    Simplify h_i to s_i
    Create new umbrella {o_i, s_i}
    Compute new umbrella vertex colors
end for
```

The up to $4k$ umbrellas are grouped into k clusters by running the k-means algorithm on the umbrella centers. In Figure 9 each of the 4 cells (left, white squares) contains up to k=8 umbrellas (leafs delimited by blue lines), also counting umbrellas that only partially overlap with a cell. The cells are merged into a single cell (right, white square) with k=8 new umbrellas (red lines). A new umbrella is constructed for each cluster. The center o_i of the new umbrella is set as the center of mass of the centers of the umbrellas in the cluster. The peripheral vertices s_i of the new umbrella are derived from the convex hull h_i of the cluster. h_i is simplified to stop the proliferation of vertices as the algorithm is run hierarchically. A maximum number of peripheral vertices is enforced by removing vertices with edge angles closest to 180°.

Once the shape of the new umbrella is known, its color is defined by computing colors for each of its vertices. New umbrellas are not texture mapped, thus they do not incur a significant additional storage cost. Figure 9 right shows the vertex colors for the new umbrellas. The color of a vertex is computed as a weighted sum of the color samples in the neighborhood of the vertex and inside the new umbrella. We use a raised cosine reconstruction filter with a base of half the distance from the vertex to the umbrella center. For the center, the base is half the distance to the peripheral vertices.

Our LoD algorithm essentially implements mipmapping directly in the grid of umbrellas. Just like in conventional mipmapping, LoDs are pre-computed offline to avoid the performance penalty of on-the-fly LoD adaptation. The algorithm resorts to two main approximations. First, the cluster of umbrellas is approximated with its convex hull. Second, the color information stored in the textures of the umbrellas is approximated using vertex colors. These approximations work well since the output image footprint of the umbrellas is small. Figure 10 (left) shows the output of our algorithm for the case shown in Figure 9. The output image footprint of the merged cell is 8×8 pixels, shown here magnified for illustration purposes. The result is comparable to mipmapping the corresponding high resolution texture (right).

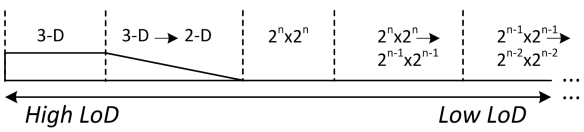

Figure 8: *Just-in-time texture synthesis LoD continuum.*

6. Just-in-time texture mapping

The synthesized texture is encoded using a 1-D array of base umbrellas, a 1-D array of modified umbrellas, and a hierarchy of 2-D arrays for the arrangement grid. A base umbrella is encoded with a texture map and with the texture coordinates of its vertices. A modified umbrella is encoded with the index of its base umbrella, with per vertex color and position, and, if 3-D shape is desired, with parameters defining the underlying ellipsoid. The arrangement grid stores an array of modified umbrella indices for each cell. This encoding is used to texture surfaces as required by the output frame.

Consider a polygon to be textured with our technique. In order to render the 3-D detail with the correct silhouette, the polygon is extruded to form a prism with height h, where h is the maximum height of the 3-D detail. Each pixel touched by the prism is textured with using Algorithm 2.

The ray at the current pixel is first intersected with the prism to find the ray segment P_0P_1 (see also Figure 11). Then

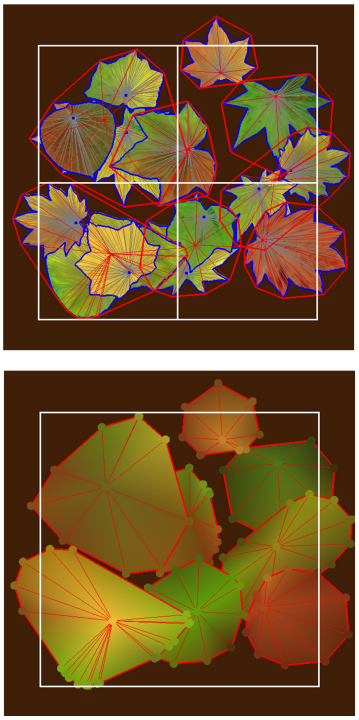

Figure 9: *Arrangement grid LoD. Four neighboring grid cells (top) are merged into one (bottom).*

P_0P_1 is traced through the 3-D grid of cells, starting from the grid cell that contains the starting point P_0. The first step in processing a cell is to set the amount of 3-D detail that has to be rendered based on the current ray. In Figure 11 the ray traverses cells c_0 to c_6. Cells c_0 and c_1 have full-height 3-D detail, the height of the 3-D detail is tapered off over cells c_2 - c_4, and then cells c_5 and c_6 have no 3-D detail (see red line). A cell with 3-D detail is intersected with the ray as described in Section 6.1. If an intersection is found, the traversal stops and the sample is returned. When no intersection is found with the current 3-D cell, the algorithm continues with the next cell traversed by the ray. The traversal terminates the first time a 2-D cell is encountered, i.e. a cell where the 3-D detail has been tapered off completely. The texture is looked up at the intersection point between the ray and the base polygon and the sample is returned. The lookup algorithm is given in Section 6.2. In our example the texture is looked up at P_1 when 2-D cell c_4 is processed.

Algorithm 2 *Per-pixel just-in-time texturing.*

```
P0P1 = pixel ray intersected with prism
cell = GetCell(P0)
while cell do
    Set3DLoD(cell)
    if cell is 3-D then
        if (S = Intersect3D(cell, P0P1)) != 0 then return S
        cell = NextCell(cell, P0P1)
        continue
    end if
    return LookUp2D(P1)                    ▷ cell is 2-D
end while
return no-sample
```

6.1. 3-D texture lookup

A grid cell with 3-D detail (*cell*) is intersected with a ray (P_0P_1) according to Algorithm 3. The ray P_0P_1 is first clipped with the axis aligned bounding box of the cell, obtaining R_0R_1. The ray segment R_0R_1 is then modified to account for the possible reduction in height of the 3-D detail. Modifying the ray and intersecting the uncompressed cell with the modified ray is easier than compressing the cell and intersecting it with the original ray. In Figure 11 the ray is not modified for cells c_0 and c_1 which are rendered with full-height 3-D detail, but ray segment ab is modified for cell c_2 to ab', by moving b to b'. b' is found such that $b'b_0$ / h = bb_0 / b_1b_0. The resulting ray ab' has the same endpoints with respect to the uncompressed cell as the original ray segment ab with respect to the compressed cell. ab' is intersected with the uncompressed cell. Similarly, bd is modified to $b'd'$, maintaining ray continuity from cell c_2 to cell c_3 (green line $ab'd'$). For cell c_4 de is above the compressed cell thus no intersection needs to be computed (dotted green line).

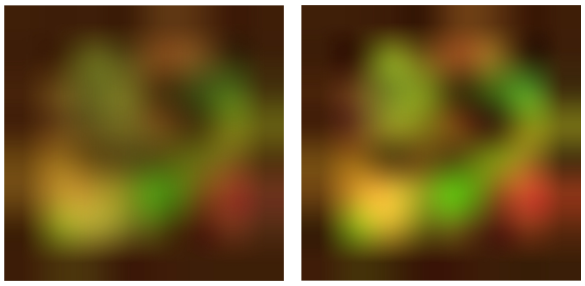

Figure 10: *Comparison between minification with our LoD algorithm (left) and with mipmapping (right).*

Algorithm 3 *Intersection of pixel ray with 3-D cell.*

```
R0R1 = ClipRayWithCellBoundingBox(P0P1, cell)
Q0Q1 = ModifyRay(R0R1, cell)
S = no-sample
for all modified umbrellas u in cell do
    if (Q = Intersect(Q0Q1, u.ellipsoid)) != 0 then
     if (Si = LookUp(Q, u.2Dpolygon)) != 0 then
      S = ClosestToEye(S, Si);
end for
return S
```

The cell is intersected with the modified ray by intersecting each umbrella in the cell, and by recording the closest intersection. An umbrella is intersected by first intersecting the ellipsoid defining its 3-D shape and then by intersecting the polygon defining its 2-D shape. The ellipsoid intersection implies solving a quadratic. The ellipsoid surface parameter values at the intersection define the 2-D point where the polygon is looked up, as described in Section 6.2.

6.2. 2-D texture lookup

Given point P on the base polygon with texture coords (s,t), the color at P is looked up with Algorithm 4. Since the grid is uniform, the cell containing P is found directly by dividing s and t by the width and height of the cell. The modified umbrellas in the cell are traversed in front to back order in search of an intersection. The base umbrella sector possibly containing P is found using the angle φ between the vector defined by P and the horizontal axis (Figure 12). If P is actually inside the triangle sector, an intersection has been found and a color is returned. The color is computed by blending

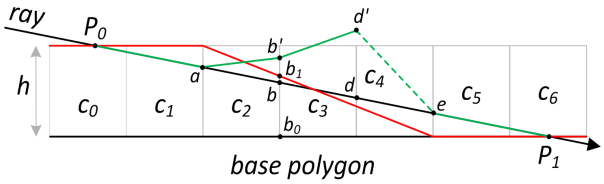

Figure 11: *Intersection between ray and grid cells ($c_0 - c_6$).*

Algorithm 4 *2-D texture lookup at point P.*

Find grid *cell* that contains P
for all modified umbrellas u in *cell* **do**
  Compute angle φ of P with the horizontal axis
  Use φ to find sector triangle T_j containing P
  if P outside peripheral edge e_j **then continue**
  Compute barycentric coords. (α, β, γ) of P in T_j
  Lookup base umbrella texture color c_t at (α, β, γ)
  Compute interpolated vertex color c_v
  return blended final color $c_t + f c_v$
end for
return background color

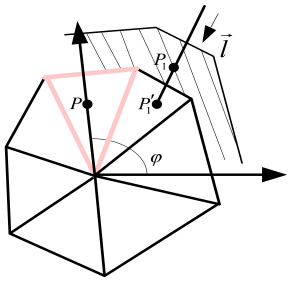

Figure 12: *Identification of umbrella triangle containing lookup point and approximate shadow computation.*

the base umbrella texture color with the interpolated color of the triangle sector (found using Equation 1). If no modified umbrella covers P, the background color is returned.

The level of detail is adapted by examining the derivatives of the texture coordinates in arrangement grid units. While these derivatives are sufficiently small (i.e. 0.125), the LoD is simply adapted by looking up the base umbrella textures with mipmapping. Once the derivatives become too large, the coarser LoDs of the arrangement grid will be used. The lookup algorithm is run on two adjacent LoDs that bracket the desired LoD, and a linear interpolation produces the final color, similarly to the trilinear interpolation of conventional mipmapping. The algorithm provides a smooth transition between LoDs (Figure 2 and video).

Figure 13: *Texture with (left) and without (right) shadows.*

Figure 14: *Additional texture synthesis examples and corresponding base umbrellas.*

We enhance the appearance of the texture by approximating shadows with a small addition to Algorithm 4 (Figure 13). As the umbrellas of the cell are traversed we also test whether the lookup point is in the shadow cast by the current umbrella. This is done by moving the point towards the light on the texture plane, and by testing whether the displaced point is inside the umbrella, which would indicate that the original point is in the umbrella's shadow. In Figure 12 P_1 is translated along the light vector l to P'_1 which is inside the umbrella thus P_1 is in shadow.

7. Results and Discussion

We have applied our technique to generate and use several textures: *Fall Leafs* (Figure 1), *Berries* (Figure 14, row 1),

Pepper (row 2), *Green Leafs* (row 3), and *Flowers* (row 4). The resolution of the base umbrella textures is 256×256, which allows zooming in with good detail. Our LoD algorithm provides quality results even at extreme minification rates. As illustrated in the video, our technique is stable which preserves quality in sequences of frames. Umbrella edges are antialiased using conventional multisampling.

7.1. Storage Reduction Performance

In order to quantify the texture memory savings brought by our just-in-time texture encoding, let's assume that there are b base umbrellas, each with v vertices and with a texture of resolution $w \times h$. The storage cost of a base umbrella is $wh + 2v$ four-byte words, where we counted 2 floats per vertex for the texture coordinates. Let n be the number of modified umbrellas. The cost of a modified umbrella is $1 + 3v$ words, which accounts for the base umbrella index and for the positions and colors of the vertices. Let us assume that the arrangement grid has a resolution of $W \times H$ and that there are at most k modified umbrellas per grid cell. Each grid cell records the modified umbrellas it stores with k integer indices. The overall cost in four-byte words J of the just-in-time texture encoding is thus

$$J = bwh + 2bv + n(1+3v) + kWH \tag{2}$$

In order to compare this cost to that of a conventional approach storing the synthesized texture explicitly, first we have to determine the resolution of the synthesized texture. Since modified umbrellas have different sizes, the resolution of the texture has to be determined by examining the resolution at individual modified umbrellas. A modified umbrella U_i with an arrangement grid axis aligned bounding box of $x_i \times y_i$ implies a synthesized texture resolution of $w/x_i \times h/y_i \times W \times H$. The dimensions x_i and y_i are measured in grid cell units. In order to not lose information at any of the modified umbrellas, the synthesized texture should have a resolution T of

$$T = max(w/x_i) \times max(h/y_i) \times W \times H \tag{3}$$

where the maxima are computed over all modified umbrellas. Using min instead of max in the equation above corresponds to a synthesized texture that loses color resolution at all modified umbrellas but the one with the largest arrangement grid footprint. A third option is to use the average resolution over all modified umbrellas. Table 1 gives the storage reduction factors achieved by our method versus conventional texture synthesis, for each of these 3 options. Just-in-time texture synthesis achieves non-lossy compression with substantial factors. Base umbrella texture resolution $w \times h$ is 256×256, arrangement grid resolution $W \times H$ is 64×24 (10×13 for *Flowers*), and the values for the other texture synthesis parameters are given in the table.

Table 1: Storage reduction performance

Texture	b	v	k	n ($\times 1{,}000$)	Min	Avg	Max
Fall L.	9	63	19	6.1	16:1	25:1	47:1
Berries	3	58	16	14	10:1	23:1	48:1
Green L.	6	51	68	25	5:1	13 :1	53 :1
Peppers	4	35	37	14	21:1	32:1	56:1
Flowers	6	74	12	.39	5:1	10:1	18:1

So far the storage cost analysis was conducted under the assumption that texel data is stored uncompressed. This precludes any loss of quality due to compression and avoids decoding costs. Texel data compression can be used with just-in-time texture synthesis by compressing the base umbrella textures. Compression benefits just-in-time texturing less than it benefits conventional texture synthesis, because the just-in-time encoding is already a compressed representation. Even so, for example for *Fall Leafs*, when both the base umbrella textures and the conventional synthesized texture are stored in jpeg format with a quality factor of 50%, just-in-time texture synthesis still achieves a storage reduction factor of 3:1.

7.2. Rendering Performance

Just-in-time texture synthesis achieves storage savings by shifting the texture expansion from pre-processing to runtime. This is a classic tradeoff between storage and computation cost. Instead of a single mipmapped lookup, the fragment program has to compute the intersection between the sampling location and the modified umbrellas at the current grid cell. As such, the rendering cost depends on two main factors: the number of modified umbrellas per grid cell k and the complexity of the umbrellas v. Figure 15 shows the variation of the rendering performance with k and v for *Fall Leafs*. The output image resolution is 512×512, $W \times H = 32 \times 32$, and $w \times h = 256 \times 256$. Rendering was done with 4x multisampling and with the approximate shadow algorithm described. The shadow algorithm brings only a small frame rate penalty of under 2 Hz. Performance was measured on an Intel Core i5-760 2.80GHz PC with an NVIDIA GeForce GTX 470, 1,280 MB graphics card.

7.3. Limitations

Even though the umbrella is a versatile geometry and color representation, not all texture elements can be modeled efficiently with umbrellas (e.g. grass blades). Another limitation is that the LoD hierarchy is built using the convex hull, which increases the size of markedly concave structures. Also in the case of umbrellas with great color variation, the color information culling from one level of the LoD hierarchy to the next could be too aggressive–there is a single color sample inside the convex hull. Finally, rendering performance of just-in-time texture synthesis is lower than

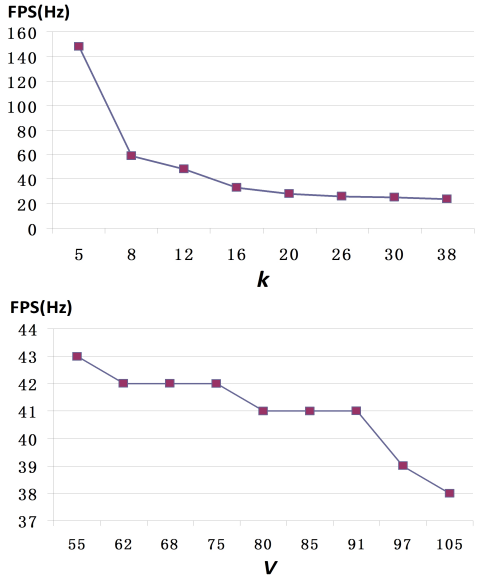

Figure 15: *Rendering performance variation with k (top, v = 77), and v (bottom, k = 13) for Fall Leafs.*

for conventional texture mapping and it decreases with the overlap factor.

8. Conclusions and Future Work

We have presented a novel texture synthesis approach that models texture patches with parameterizable color, 2-D, and 3-D shape. A few input patches are sufficient to mimic the diversity present in nature. Texel data is computed just-in-time, which brings substantial storage savings. An LoD scheme only renders 3-D detail where needed, and supports artifact-free minification at any level.

One possible direction of future work is to alleviate some of the limitations discussed above. The 2-D shape and color modeling power of umbrellas can be further increased by introducing additional vertices on the radii connecting the center to the peripheral vertices. This would allow for greater color diversification and LoD adaptation flexibility. The LoD shape fidelity could be improved by not requiring that the merged umbrella be convex, but rather by shrink wrapping it to the actual perimeter of the umbrellas in the cluster it replaces. The 3-D modeling power of our technique could be increased by mapping the umbrella to a more flexible 3-D representation such as a height field. The high level algorithm would remain the same, including the scheme for smoothly transitioning from 3-D to 2-D, with the only change of replacing the ray/ellipsoid intersection with a more expensive iterative ray/height field intersection.

Our paper focuses on the texture synthesis sub-problems of diversification, LoD adaptation, and run-time lookup. Another possible direction of future work is to integrate our method with prior solutions to other texture synthesis sub-problems such as automatic extraction of texture elements and of arrangement patterns, and such as distortion-free tiling on 2-D and 3-D domains.

Just-in-time texture synthesis takes advantage of the programmability sophistication of modern graphics hardware by replacing the texel with a higher-level texturing primitive. Our method brings benefits whose importance will only grow as increases in computation performance continue to outpace increases in storage and bandwidth.

References

[BAC96] BEERS A., AGRAWALA M., CHADDHA N.: Rendering from compressed textures. In *Proceedings of the ACM SIGGRAPH '96* (1996), pp. 373–378. 3

[BIP00] BAJAJ C., IHM I., PARK S. H.: Compression-based 3D texture mapping for real-time rendering. In *Graphical Models-Pacific Graphics '99* (2000), vol. 62, pp. 391–410. 3

[Bli78] BLINN J.: Simulation of wrinkled surface. *5th annual conference on Computer Graphics and Interactive Techniques, ACM Press* (1978), 286–292. 3

[Coo84] COOK R.: Interactive horizon mapping. *11th annual conference on Computer graphics and interactive techniques, ACM Press* (1984), 223–231. 3

[CSHD03] COHEN M. F., SHADE J., HILLER S., DEUSSEN O.: Wang tiles for image and texture generation. *ACM Transactions on Graphics 22*, 3 (2003), 287–295. SIGGRAPH 2003. 2, 3

[DCH05] DIVERDI S., CANDUSSI N., HOLLERER T.: *Real-time rendering with wavelet-compressed multi-dimensional textures on the GPU*. Computer Science Technical Report 2005-05, 2005. 3

[DGAG06] DESBENOIT B., GALIN E., AKKOUCHE S., GROSJEAN J.: Modeling autumn sceneries. In *Eurographics(EG) short paper* (2006). 3

[DLC05] DONG F., LIN H., CLAPWORTHY G.: Cutting and pasting irregularly shaped patches for texture synthesis. In *Computer Graphics Forum* (2005), vol. 24, pp. 17–26. 2

[DMLG02] DISCHLER J. M., MARITAUD K., LEVY B., GHAZANFARPOUR D.: Texture particles. *Computer Graphics Forum 21*, 3 (Sept. 2002), 401–410. 2

[DZ06] DISCHLER J. M., ZARA F.: Real-time structured texture synthesis and editing using image-mesh analogies. In *The Visual Computer* (2006), vol. 22, pp. 926–935. 2

[EF01] EFROS A. A., FREEMAN W. T.: Image quilting for texture synthesis and transfer. In *SIGGRAPH 2001, Computer Graphics Proceedings* (2001), pp. 341–346. 2

[Gla04] GLANVILLE S.: Texture bombing. In *GPU Gems chapter 20* (2004). http://http.developer.nvidia.com/GPUGems/gpugems_ch20.html. 3

[Gum03] GUMHOLD S.: Splatting illuminated ellipsoids with depth correction. *8th International Fall Workshop on Vision, Modelling and Visualization* (2003), 245–252. 3

[HDKS00] HEIDRICH W., DAUBERT K., KAUTZ J., SEIDEL H.: Illumination micro geometry based on precomputed visibility. In *SIGGRAPH 2000, Computer Graphics* (2000), pp. 455–464. 3

[HQXT05] HUANG J., QI D., XIONG C., TANG Z.: Foreground-distortion method for image synthesis. In *Proceedings of the 9th International Conference on Computer Aided Design and Computer Graphics* (2005), pp. 509–513. 2

[IMIM08] IJIRI T., MECH R., IGARASHI T., MILLEI G.: An example-based procedural system for element arrangement. In *Computer Graphics Forum* (2008), vol. 27, pp. 429–436. 2

[KCoDL06] KOPF J., COHEN-OR D., DEUSSEN O., LISCHINSKI D.: Recursive wang tiles for real-time blue noise. *ACM Transactions on Graphics 25*, 3 (2006), 509–518. (Proceedings of SIGGRAPH 2006). 2

[KE02] KRAUS M., ERTL T.: Adaptive texture maps. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware* (2002), pp. 1–10. 3

[KS01] KAUTZ J., SELDEL H.: Hardware accelerated displacement mapping for image based rendering. *Graphics Interface* (2001), 61–70. 3

[KSE*03] KWATRA V., SCHODL A., ESSA I., TURK G., BOBICK A.: Graphcut textures : Image and video synthesis using graph cuts. *ACM TOG 22*, 3 (July 2003), 277–286. 2

[KTI*01] KANEKO T., TAKAHEL T., INAMI M., KAWAKAMI N., YANAGIDA Y., MAEDA T., TACHI S.: Detailed shape representation with parallax mapping. *ICAT 2001* (2001), 205–208. 3

[LD05] LAGAE A., DUTRE P.: A procedural object distribution function. In *ACM Transactions on Graphics* (2005), vol. 24, pp. 1442–1461. 3

[LH06] LEFEBVRE S., HOPPE H.: Appearance-space texture synthesis. *ACM TOG 25*, 3 (July 2006), 541–548. (SIGGRAPH 2006). 2

[LHN05] LEFEBVER S., HORNUS S., NEYERT F.: Texture sprites: Texture elements splatted on surfaces. In *ACM-SIGGRAPH Symposium on Interactive 3D Graphics* (2005). 3

[LLH04] LIU Y. X., LIN W. C., HAYS J. H.: Near-regular texture analysis and manipulation. *ACM TOG 23*, 3 (Aug. 2004), 368–376. (SIGGRAPH 2004). 2

[LLX*01] LIANG L., LIU C., XU Y.-Q., GUO B., SHUM H.-Y.: Real-time texture synthesis by patch-based sampling. *ACM TOG 20*, 3 (July 2001), 127–150. 2

[LN03] LEFEBVRE S., NEYRET F.: Pattern based procedural textures. In *ACM-SIGGRAPH Symposium on Interactive 3D Graphics* (2003), pp. 203–212. 2, 3

[LP00] LEFEBVRE L., POULIN P.: Analysis and synthesis of structural textures. In *Graphics Interface* (2000), pp. 77–86. 2

[LTcL05] LIU Y., TSIN Y., CHIEH LIN W.: The promise and perils of near-regular texture. *International Journal of Computer Vision- Special Issue on Texture Analysis and Synthesis 62*, 1-2 (April-May 2005), 145–159. 2

[Max88] MAX N.: Horizon mapping: Shadows for bump-mapped surfaces. In *The Visual Computer* (1988), vol. 4, pp. 109–117. 3

[MGGA10] MARECHAL N., GALIN E., GUERIN E., AKKOUCHE S.: Component-based model synthesis for low polygonal models. In *Proceedings of Graphics Interface 2010* (2010), pp. 217–224. 3

[MWT11] MA C. Y., WEI L. Y., TONG X.: Discrete element textures. *ACM Transactions on Graphics 30*, 4 (2011), 62:1–10. (SIGGRAPH 2011). 2

[NMMK05] NICOLL A., MESETH J., MULLER G., KLEIN R.: Fractional fourier texture masks: Guiding near-regular texture synthesis. *Computer Graphics Forum 24*, 3 (Sept. 2005), 569–579. 2

[Pag04] PAGET R.: Strong markov random field model. *IEEE Transactions on Pattern Analysis and Machine Intelligence 26*, 3 (2004), 408–413. 2

[PFH00] PRAUN E., FINKELSTEIN A., HOPPE H.: Lapped textures. *27th annual conference on Computer graphics and interactive techniques* (2000), 465–470. 2, 3

[POC05] POLICARPO F., OLIVEIRA M., COMBA J.: Real-time relief mapping on arbitrary polygonal surfaces. In *ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games* (2005), pp. 155–162. 3

[PTMG08] PEVRAT A., TERRAZ O., MERILLOU S., GALIN E.: Generation vast varieties of realistic leaves with parametric 2gmaps l-systems. In *The Visual Computer* (2008), vol. 24, pp. 807–816. 3

[RCS04] RODKAEW Y., CHONGSTITVATANA P., SIRIPANT S.: Modeling plant leaves in marble-patterned colours with particle transportation system. In *Proceedings of the 4th International Workshop on Functional-Structural Plant Models* (2004), pp. 391–397. 3

[RHDG10] RISSER E., HAN C., DAHYOT R., GRINSPUN E.: Synthesizing structured image hybrids. *ACM TOG 29*, 4 (2010), 85:1–85:6. 2

[RSL*02] RODKAEW Y., SIRIPANT S., LURSINSAP C., CHONGSTITVATANA P., FUJIMOTO T., N. C.: Modeling leaf shapes using l-system and genetic algorithms. In *In International Conference NICOGRAPH (April)* (2002), pp. 73–88. 3

[SA79] SCHACHTER B., AHUJA N.: Random pattern generation processes. In *Computer Graphics Forum* (1979), vol. 10, pp. 95–114. 2

[SC00] SLOAN P., COHEN M.: Interactive horizon mapping. *Eurographics Workshop on Rendering Techniques* (2000), 281–286. 3

[SR06] STACHERA J., ROKITA P.: Gpu-based hierarchical texture decompression. In *Eurographics '06* (2006). 3

[STC09] SUN C. H., TSAO Y. M., CHIEN S. Y.: High-quality mipmapping texture compression with alpha maps for graphics processing units. In *IEEE Transactions on Multimedia* (2009), vol. 11, pp. 589–599. 3

[TF08] TANG Y., FAN J.: Incremental texture compression for real-time rendering. In *Proceedings of the 4th International Symposium on Advances in Visual Computing (ISVC '08), Part II* (2008), vol. 5359, pp. 1076–1085. 3

[TW08] TZENG S., WEI L.: Parallel white noise generation on a gpu via cryptographic hash. *2008 symposium on Interactive 3D graphics and games* (2008). 2

[Wei04] WEI L.: Tile-based texture mapping on graphics hardware. *SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (2004), 55–63. 3

[Wil83] WILLIAMS L.: Pyramidal parametrics. *SIGGRAPH '83, Proceedings of the 10th annual conference on Computer graphics and interactive techniques 17*, 3 (1983). 3

[WLKT09] WEI L.-Y., LEFEBVRE S., KWATRA V., TURK G.: State of the art in example-based texture synthesis. In *Eurographics 2009, State of the Art Report, EG-STAR* (2009). 2

[WWT*03] WANG L., WANG X., TONG X., LIN S., HU S., GUO B., SHUM H.: View-dependent displacement mapping. In *ACM Trans, Graph* (2003), vol. 22, pp. 334–339. 3

[WY04] WU Q., YU Y. Z.: Feature matching and deformation for texture synthesis. *ACM TOG 23*, 3 (Aug. 2004), 362–365. 2

[ZZV*03] ZHANG J., ZHOU K., VELHO L., GUO B., SHUM H.: Synthesis of progressively-variant textures on arbitrary surfaces. *ACM TOG 22*, 3 (July 2003), 295–302. (SIGGRAPH '03). 2