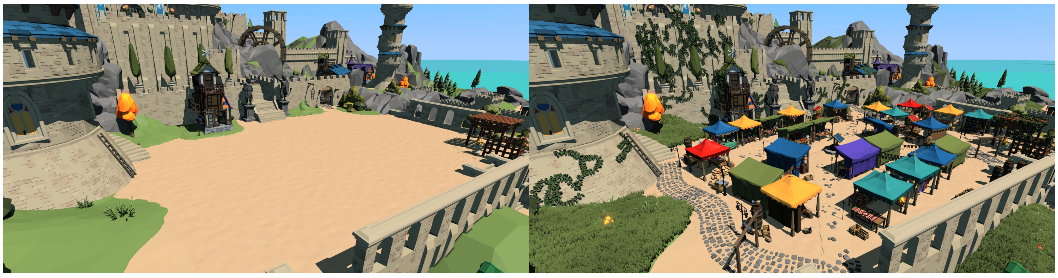
Real-Time Procedural Generation with GPU Work Graphs

BASTIAN KUTH, Coburg University, Germany
MAX OBERBERGER, AMD, Germany
CARSTEN FABER, Coburg University, Germany
DOMINIK BAUMEISTER, AMD, Germany
MATTHÄUS CHAJDAS, AMD, Germany
QUIRIN MEYER, Coburg University, Germany



(a) Without our system (b) With our system

Fig. 1. Our GPU-only work-graphs system uses 37 nodes to augment an existing scene (a) in real-time. All new elements in (b) are generated by our system each frame, including the marketplace, the ivy on the walls, paths, and the grass.

We present a system for real-time procedural generation that makes use of the novel GPU programming model, *work graphs*. The nodes of a work graph are shaders, which dynamically generate new workloads for connected nodes. This greatly simplifies the implementation of recursive procedural algorithms on GPUs. Combined with GPU ray tracing and procedural mesh shaders, our system makes use of this graph structure to tackle various common problems of procedural generation. Our system is very easy to implement, requiring no additional data structures from what would already be available in a modern rendering engine. We demonstrate the real-time editing capabilities on representative examples. We augment the scene in the teaser image with 79,710 instances in 3.74 ms on an AMD Radeon RX 7900 XTX.

CCS Concepts: • **Computing methodologies** → **Mesh models**; *Massively parallel algorithms.*

Additional Key Words and Phrases: work graphs, geometry generation, ray tracing, mesh shaders

ACM Reference Format:
Bastian Kuth, Max Oberberger, Carsten Faber, Dominik Baumeister, Matthäus Chajdas, and Quirin Meyer. 2024. Real-Time Procedural Generation with GPU Work Graphs. *Proc. ACM Comput. Graph. Interact. Tech.* 7, 3 (July 2024), 16 pages. https://doi.org/10.1145/3675376

Corresponding author: Bastian Kuth, Coburg University, Germany.

1 INTRODUCTION

Creating highly detailed virtual worlds manually is a labor-intensive task. Procedural design tools like Blender [Blender Online Community 2024] or Houdini [Side Effects Software Inc. 2023] accelerate this process. The generated geometry, however, is typically *baked* to a polygonal format. While this allows for simple game-engine integration, it comes with multiple disadvantages:

- *Data Duplication:* For rendering, all baked data has to be stored on disk, loaded by the CPU, and streamed to the GPU.
- *Tool Disparity:* Scenes in the design tools usually look and behave differently than in the game-engine. Hence, artists must reiterate through, often several, tools during design.
- *Scene Staticity:* The baked data is static and does not allow for the end user to modify a scene. Baked geometry can only react in a superficial way to dynamic user input, e.g., by disappearing when in the way of a user-placed object.

Therefore, the generation process is migrating into game engines [Epic Games 2024]. Since most algorithms are CPU-based, data duplication remains unsolved. Moreover, many techniques are too slow to run every frame. This prevents real-time interaction, degrades artist productivity, and even misses out on optimization opportunities, like creating only content relevant for the next frame. Many graphics problems are accelerated by GPU implementations, unfortunately, generation algorithms fail to map well to current GPU programming models.

Recently, *GPU work graphs* expose a newly introduced programming model for real-time graphics [Microsoft Cooperation 2024]. It enables GPU workloads, thus shader invocations, to generate and launch other GPU workloads. Work graphs improve GPU programmability by enabling multi-level work amplification with dynamic workloads. To our knowledge, we present the first GPU work-graphs-based procedural generation and rendering system that creates various geometry types in real-time. Thereby, we make the following contributions:

- We provide a *work-graphs overview*, as we are not aware of any scientific publication using them.
- We show that *GPU ray tracing* is a powerful and convenient tool to achieve dependent generation.
- We utilize *mesh shaders* for procedural mesh generation.
- We provide an algorithm for work graphs to achieve *instancing*.
- With a recursive work-graph node, we dynamically *generate ivy* on existing geometry.
- With work graphs, we solve the *straight skeleton of a polygon* problem to generate a *marketplace*.
- We utilize our system to generate *frustum-culled ground clutter*, such as grass, flowers, and insects.

Our system generates geometry in real-time, is extensible, and easy to implement.

2 RELATED WORK

Following Direct3D12's conventions, we call a GPU program *shader*. It is executed using *threads* clustered to *thread groups*. A *wave* subdivides a thread group into typically 32 threads that run on a *single instruction, multiple data (SIMD) unit* in lockstep. *Wave intrinsics* are instructions for fast communication within a wave, *group shared memory* is used to communicate between waves.

Work graphs tackle the problem of *GPU dynamic work creation and processing*. It enables many graphics applications like ray tracing [Parker et al. 2010], REYES rendering [Steinberger et al. 2014a], geometry generation [Steinberger et al. 2014b], vector graphics [Dokter et al. 2019], rasterization [Patney et al. 2015], and implicit surface rendering [Jazar and Kry 2023].

One way to motivate dynamic work creation for GPU real-time rendering is *work amplification*. For example, an object outside the view frustum should be culled. If the object is visible, a single GPU thread amplifies to a much larger number of threads to further process the object. A second reason for dynamic work creation is to avoid *thread divergence*. Execution performance of threads in the same SIMD unit degrades when they take different branches in the control flow. A work creation system may cluster new work items of the same branch to run on the same SIMD unit.

Persistent threads [Aila and Laine 2009] pioneered dynamic GPU work processing. Later, Tzeng et al. [2010] add dynamic work creation, Softshell [Steinberger et al. 2012] supports multiple task types, and Whippletree [Steinberger et al. 2014a] introduces several optimizations. Orr et al. [2014] demonstrate an abstraction on data-parallel hardware where algorithms are modeled with flow graphs [Gaster and Howes 2012]. However, a reliable and efficient implementation requires deep understanding about GPU hardware, scheduling, memory management [Winter et al. 2021], and data-parallel queues [Kenzel et al. 2023; Kerbl et al. 2018, 2017; Scogland and Feng 2015]. CUDA Graph API allows to better schedule dependent kernel launches from the CPU. CUDA Dynamic Parallelism (CDP) enables a GPU thread to launch a full thread grid [NVIDIA 2024]. Unlike work graphs, this makes the developer responsible for fine work scheduling. Therefore, CDP may perform poorly on small grids [Kerbl et al. 2022] and is only supported by one vendor.

In practice, *indirect execution* is a fairly common technique for dynamic work creation due to its widespread support. With indirect execution, the CPU still controls which kind of work is executed in what order, but the amount of work is defined by parameters in GPU memory. Therefore, work amplification splits into two isolated stages: In the *creation* stage, GPU threads create a new work item by incrementing a counter per work-item type with optional parameters stored in a buffer. In the *execution* stage, the GPU uses the counter to dispatch as many work items as the creation stage requested. For frustum culling, the creation stage writes what objects to draw and the execution stage performs the rendering. The widespread support is shadowed by several limitations:

- *Worst Case Memory Allocation.* An indirect execution system has to provide enough memory to hold parameters of all work items that could be created in the creation stage. For frustum culling, the buffer containing the draw command information needs to be large enough to hold the parameters for drawing all scene objects that can be visible at the same time.
- *Barrier Synchronization.* For an indirect execution system to work, there must be a barrier between the two stages. This stalls the GPU for a brief moment and therefore degrades performance.
- *Convoluted Implementation.* The implementation of such a system can become convoluted as soon as the execution stage also creates new work items. Here, a developer must keep track of all possible execution paths, the worst case memory requirements, and synchronization.

To mitigate the limitations of previous techniques, GPU work graphs were introduced to Direct3D12 [Microsoft Cooperation 2024] and Vulkan [Hector et al. 2023].

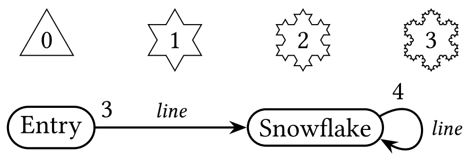
Since the topic of *procedural geometry generation* is vast, we limit the discussion related to the type of models we generate. *Ivy generation* falls into the realm of plant generation pioneered by L-Systems [Prusinkiewicz and Lindenmayer 1990], for which a data-parallel evolver exists [Lipp et al. 2010]. Shape grammars are another approach to plant generation [Stiny and Gips 1971]. Marvie et al. [2012] propose a GPU system procedurally creating geometry using shape grammars. Buron et al. [2015] build upon that system to generate ivy. They reach interactive frame-rates with hardware tessellation and geometry shaders. To generate our *market layout*, we utilize the straight skeleton of a polygon [Aichholzer et al. 1996] which is also used as a city parcel generation method [Vanegas et al. 2012]. In procedural modeling, the skeleton is used for building roofs [Brenner 2000; Merrell et al. 2010; Müller et al. 2006]. Other use cases include polygon morphing [Barequet and Yakersberg 2003] or path planning [Oksanen and Visala 2009]. Our market layout is similar to city planning [Parish and Müller 2001], but see Vanegas et al. [2010] for an overview. Grass modeling and rendering is a common problem from procedural generation [Boulanger et al. 2009; Fan et al. 2015; Jahrmann and Wimmer 2013]. Papavasiliou [2015] dynamically generates different kinds of ground clutter on a terrain surface using tessellation and geometry shaders.

3 GPU WORK GRAPHS

A *work graph* [Microsoft Cooperation 2024] is a directed graph with a maximum depth of 32. Its *nodes* are compute shaders. Its *edges* denote which nodes can create work for other nodes. A *record* refers to a single entity of work, optionally parameterized with a *record struct*. A work graph is entered via *entry nodes*. *Graphic leaf nodes* for geometry rasterization are specified, but not yet available in a public driver. A node is associated with one of three *launch modes*:

- *The broadcasting launch* dispatches a 3D grid of thread groups, similar to a compute shader dispatch. All thread groups receive the same input record. The number of thread groups launched can optionally be dynamically determined as part of the record, while the thread group size is static. This launch mode is suitable for massive parallel workloads. Wave intrinsics and group-shared memory allow for low cost communication between the threads of a work group.
- *The thread launch* is for workloads that only require a single isolated thread. Thus, there is one record per thread. For each thread-wise launched node, the GPU scheduler can bundle multiple records until a sufficient amount is reached, ideally when a full SIMD lane can be occupied or when no more work items of that type can be produced.
- *The coalescing launch* extends the thread launch by exposing the number of records provided to a thread group to the developer. The thread group cooperatively processes the variable number of input records. However, the work-graphs specification does not guarantee any bundling.

We provide a toy example for generating the Koch snowflake fractal with work graphs:



```
struct Line { float2 a, b; };
[Shader("node")]
[NodeLaunch("thread")]
[NodeIsProgramEntry]
void Entry([MaxRecords(3)] NodeOutput<Line> Snowflake) {
    ThreadNodeOutputRecords<Line> o =
        Snowflake.GetThreadNodeOutputRecords(3);
    o.Get(0).a = o.Get(2).b = float2(1., 0.);
    o.Get(0).b = o.Get(1).a = float2(-.5, -sqrt(3) * .5);
    o.Get(1).b = o.Get(2).a = float2(-.5, +sqrt(3) * .5);
    o.OutputComplete();
}
```

```
[Shader("node")]
[NodeLaunch("thread")]
[NodeMaxRecursionDepth(3)]
void Snowflake(ThreadNodeInputRecord<Line> record,
               [MaxRecords(4)] NodeOutput<Line> Snowflake) {
    float2 a = record.Get().a, b = record.Get().b;
    bool hasOutput = GetRemainingRecursionLevels() != 0;
    ThreadNodeOutputRecords<Line> o =
        Snowflake.GetThreadNodeOutputRecords(hasOutput * 4);

    if (hasOutput) {
        float2 perp = float2(a.y - b.y, b.x - a.x) * sqrt(3) / 6;
        o.Get(0).a = a;
        o.Get(0).b = o.Get(1).a = lerp(a, b, 1./3.);
        o.Get(1).b = o.Get(2).a = lerp(a, b, .5) + perp;
        o.Get(2).b = o.Get(3).a = lerp(a, b, 2./3.);
        o.Get(3).b = b;
    } else { DrawLine(a, b); }
    o.OutputComplete();
}
```

The Entry node outputs three *Line* records, forming the initial triangle to the recursing Snowflake node. When visualizing parts of a work graph, we denote the maximum number of output records at the start of edges and the record struct at the edge center.

4 OUR GENERATION SYSTEM

Our system's *inputs* are *control parameters*. Control parameters consist of anything artists want to alter during scene design e.g., a point, a line like in our snowflake example, a bounding volume, a control cage, curve parameters, a threshold value, or a seed for random number generation. Our system uses the control parameters to dynamically generate geometry for rendering. This is similar to tessellation, where a parametric surface is dynamically evaluated based on the camera view.

Nodes of a work graph consume these inputs. When a node is launched, it uses the provided control parameters to create new work records, geometry, or both. Consider the example of a car generated by a graph: A parent node representing the entire car has a child node for each wheel.

The wheels produce geometry for the rubber tire and in turn, create child nodes that introduce additional geometric details like nuts or rims. Our system *outputs* geometry in various ways:

- *Draw Instance List.* Nodes append draw commands to a *draw instance list.* A command contains a predefined index to an *asset*, i.e., multiple meshes with multiple materials. Instead of a list, we could also write a scene graph directly from the given hierarchy.
- *Ray-tracing Instance List.* GPU ray tracing is becoming increasingly more common for lighting. Therefore, our nodes append to a *ray-tracing instance list* from which the rendering API create a top-level acceleration structure (TLAS) of a bounding volume hierarchy (BVH). Furthermore, we propose using the BVH to adapt our newly generated geometry to preexisting geometry.
- *Procedural Mesh.* Besides outputting newly arranged predefined assets with draw instance lists, the system can also output fully procedural geometry made out of vertices and triangles. Mesh shaders can forward procedural geometry to the rasterization stage directly using graphic leaf nodes. Until public drivers support graphic leaf nodes, we emulate them with indirect execution. Alternatively, the system writes a vertex and index buffer to a pre-allocated buffer. This is also required for generating a bottom-level acceleration structure (BLAS) for procedural meshes.

4.1 Coalesced Instancing with Work Graphs

Instancing is a common technique to improve rendering performance: to reduce state changes and repeated data fetching when drawing multiple instances of the same geometry, the instances are bundled together and issued in a single draw call. We contribute two concepts to collect instances of the same asset coming from different nodes both utilizing the coalescing launch mode.

Per-Asset Coalescing. Per instantiable asset X, we create a node $\boxed{\text{CoalesceAsset}[X]}$ in coalescing launch mode that bundles draw calls to X into an instanced draw call. The node then issues the combined, instanced draw call to $\boxed{\text{DrawAsset}}$, where T_i is an instance transformation of X:

$$\begin{tikzcd}
\bigcirc \arrow[dr, "T_0"'] & \bigcirc \arrow[d, "T_1"'] & \bigcirc \arrow[dl, "T_2"] & \\
& \boxed{\text{CoalesceAsset}[X]} \arrow[rr, "(T_0, T_1, T_2)"] & & \boxed{\text{DrawAsset}}
\end{tikzcd}.$$

$\boxed{\text{DrawAsset}}$ is a write to the draw instance list or a graphic leaf node, once available. Note that, when a GPU scheduler fails to coalesce sufficiently, functionality is not restricted.

Common to all $\boxed{\text{CoalesceAsset}[X]}$ is their input record type. *Node arrays* are a work-graphs feature specifically designed for such situations to improve scheduling and simplify implementation.

For a thread-launch node $\boxed{\text{A}}$, we can save the launch of $\boxed{\text{CoalesceAsset}[X]}$: We make use of the fact that the coalescing launch mode is a super-set of the thread launch mode. Therefore, we just launch $\boxed{\text{A}}$ in coalesced mode and have it issue combined draw records to $\boxed{\text{DrawAsset}}$ directly.

All-Asset Coalescing. Instead of one node per asset, we propose a single coalescing node for all assets. The thread group bundles input draw calls by asset to output one draw call per unique asset.

Depending on the exact use-case, one or the other concept, or a mixture between, are viable. To simplify our implementation, we utilize a single coalescing node for all assets.

4.2 BVH Markers

To generate geometry on top of existing procedural geometry and to allow for information exchange between nodes, our system requires a suitable data structure. We propose to utilize a GPU-ray-tracing BVH for this, which already comes with the graphics API. While this may seem counter-intuitive, this has two major advantages: First, creating, updating, and accessing the BVH is a matter of issuing API calls. Therefore, it drastically simplifies the implementation of our system. Second, hardware is optimized to efficiently handle millions of rays per frame.

Besides visible geometry, the graph can add invisible *markers* to the BVH. A marker can be any ray-traceable geometry, but is typically a primitive like a plane, bounding box, or sphere constructed from a triangle mesh. To make markers invisible for shading, we assign specific *instance masking bit flags* which are already part of today's BVHs. We use this feature to trace a subset of invisible markers only, a subset of geometry only, or any combination of both.

4.3 Generation Phases

To make use of a newly generated BVH containing procedural geometry and markers, we need to split the generation into multiple *phases*. One way to implement this is to dispatch a work graph for the first phase, and then to wait until it has finished executing. Before launching the graph for the next phase, the BVH is rebuilt. Frequently rebuilding a BVH is a common requirement for animated scenes, and thus hardware vendors aim to do this efficiently. Next, a dedicated work graph for the second phase is dispatched. Note that the work-graphs specification [Microsoft Cooperation 2024] discusses future features for pausing the execution of a graph and waiting for an event. This is likely to simplify the implementation of our system.

As an alternative to explicit phases, it is possible to only employ one graph execution and BVH rebuild per frame, reducing implementation complexity. To distinguish phases, we encode each phase with a unique *phase flag* and store them in the BVH instance mask bits. To make this work, there has to be at least one BVH flag for each generation phase, so that a phase only can hit the geometry of previous phases. This comes with the restriction that updates of later generation phases lag one or more frames behind when editing the control parameters. As we found this update delay to be barely noticeable at real-time frame rates, we use it in our implementation.

5 IVY

For the first example, we use our system to generate ivy on top of existing geometry. This demonstrates two aspects: node recursion, when the ivy branches into multiple strands, and the use of ray tracing for procedural generation on top of existing geometry. In cases where the ivy should generate on top of generated geometry, the respective phase flags have to be used. The ivy generation nodes place two different assets, one for an ivy stem and one for a leaf.

Branch. In its most simple form, the control parameter of an $\boxed{\text{IvyBranch}}$ is defined by a transformation, thus a position $\vec{p}$ and a rotation. $\vec{p}$ defines the start of the branch and the rotation contains the forward vector $\vec{f}$ denoting the growth direction, as well as the down vector $\vec{d}$ pointing towards the surface the ivy is growing on. We launch the $\boxed{\text{IvyBranch}}$ in broadcasting mode with

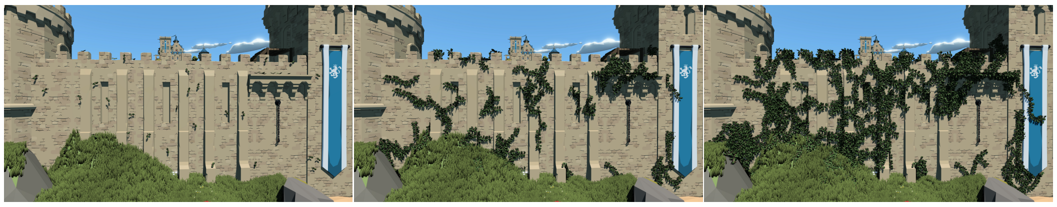

(a) 1 (b) 20 (c) 50

Fig. 2. Ivy Steps. We allow ivy to grow a maximum amount of steps. Thus, (a) shows the initial input transformations for multiple ivy strands. (b) and (c) show the generation for higher ivy length limits.

a fixed group grid size of $(1, 1, 1)$, i.e., a single thread group. We use its 32 threads of a SIMD unit to cooperatively shoot multiple rays and use wave intrinsics to find the closest hit. The IvyBranch runs the following algorithm, where b is the length of the stem asset, r the distance to the ray hit, and s the maximum user specified distance of the ivy from the surface it grows on:

```
for $n$ times do
    Query ray into $\vec{f}$ with length $b$.
    if there is a hit at $r$ then
        Draw the stem asset scaled to a length of $r$.
        if $r > \frac{b}{2}$ then
            Add leaves to the stem.
        end if
        Set the transformation to the hit position.
        Set $\vec{d}$ to the inverted surface normal.
        Set $\vec{f}$ randomly perpendicular to $\vec{d}$.
    else
        Draw the stem and leaves.
        Set the transformation to where the ray ended.
        Query ray with length $s$ in direction $\vec{d}$ with thread 0,
        and into random directions with threads 1..31.
        if $\vec{d}$ was a hit then
            Adjust $\vec{d}$ according to the surface normal.
            Randomly mutate $\vec{f}$ according to the new $\vec{d}$.
        else if thread 1..31 found new surface then
            Bias transformation towards closest hit position.
        else
            Bias transformation towards gravity vector.
        end if
    end if
    if random event and not branched yet then
        Launch child with $\vec{f}$ rotated clockwise around $\vec{d}$.
        Rotate own forward counter-clockwise around $\vec{d}$.
    end if
end for
if random continue event and recursion depth left then
    Recurse with current transformation.
end if
```

We find $n = 4$ a good trade-off: For lower n, the node needs to launch itself more often and the work-graph overhead becomes more apparent. For higher n, the output requires more memory.

Volume. To spawn multiple ivy branches at once, e.g., to cover a large surface with ivy, we introduce the IvyVolume node as a parent to a recursing IvyBranch node:

$$\begin{tikzcd}
\boxed{\text{IvyVolume}} \arrow[r, "{(\vec{p}, \vec{f}, \vec{d})}"] & \boxed{\text{IvyBranch}} \arrow[loop right, "2", "{(\vec{p}, \vec{f}, \vec{d})}"']
\end{tikzcd}$$

.

IvyVolume receives an oriented box and a density as a control parameter. In our implementation, the box only seeds the ivy, but can outgrow it. To find start points for ivy branches, IvyVolume randomly traces rays inside the box to find surfaces the ivy can grow on. The number of rays is defined by the size of the bounding box and the density input value. Their direction is defined by the orientation of the bounding box, e.g., from top to bottom. For all rays that hit a suitable surface, we launch an IvyBranch. The ivies of Fig. 8d and Fig. 2 were generated by placing a single IvyVolume.

Extensions. To extend our ivy generation algorithm, we propose the following extensions:
Generation Culling. To only generate ivy when it has a chance to be visible, we require a culling bounding volume. We suggest to use a bounding sphere for this. The position of the sphere is the initial position of the branch. To control the growth of the ivy, the user provides a maximum number of segments the IvyBranch recursion chain is allowed to spawn, see Fig. 2. This determines the worst-case radius. We use the worse-case radius to implement culling by making each individual IvyBranch terminate early when its bounding sphere is outside the camera frustum.
Growth Bias. To provide additional artist control, we add a growth bias position to the control parameters. Here, whenever the algorithm uses random values to adjust its transformation, the random values get biased towards the desired growth direction.

Blocking Material. We flag specific areas and materials in the BVH to prevent ivy from growing onto them. When GPU ray tracing in the generation hits a surface, it checks for that flag, and the generation reacts accordingly.

6 MARKETPLACE

To build a marketplace, the $\boxed{\text{Market}}$ node launches $\boxed{\text{BoothIsland}}$ nodes which surround a $\boxed{\text{Center}}$ node. A booth island gets separated from other islands with paths and consists of multiple booths with fitting props. The $\boxed{\text{Market}}$ node recursively launches itself until the remaining area is filled. The node graph looks like the following:

$$\begin{tikzcd}
 & \boxed{\text{Market}} \arrow[loop left, "2", "\textit{polygon}"'] \arrow[dl, "32"', "(\textit{line, tier})"] \arrow[d, "1", "\textit{polygon}"'] \arrow[rr, "32", "\textit{quad}"] & & \boxed{\text{BoothIsland}} \arrow[d, "\textit{transformation}"] & \\
\boxed{\text{Path}} & \boxed{\text{Center}} & & \boxed{\text{Booth}} \arrow[r, "\textit{transformation}"] & \boxed{\text{GarlandAnchor}}
\end{tikzcd}.$$

To define the shape of the marketplace, the $\boxed{\text{Market}}$ node receives a simple polygon on a terrain surface as a control parameter. We use a modified version of the *straight skeleton* [Aichholzer et al. 1996] of the polygon to create the market layout. The skeleton lines define where paths go in between the booths, as shown in Fig. 3.

A polygon's straight skeleton gets generated by shrinking all edges parallel to themselves at constant speed. During the shrinking process, the number of edges of the polygon decreases until it is fully collapsed. The straight skeleton is then given by the trajectory that the points of the polygon take while shrinking. We place paths along the straight skeleton. Additional paths are placed in parallel to the edges of the polygon to separate *rings* of booth islands. When the polygon becomes too small for additional rings, we terminate the generation by launching a $\boxed{\text{Center}}$ node that fills the remaining space with a fitting asset like a well. See Fig. 4 for an overview.

Events. During shrinking, the polygon must remain simple. Therefore, our straight skeleton algorithm handles two events:

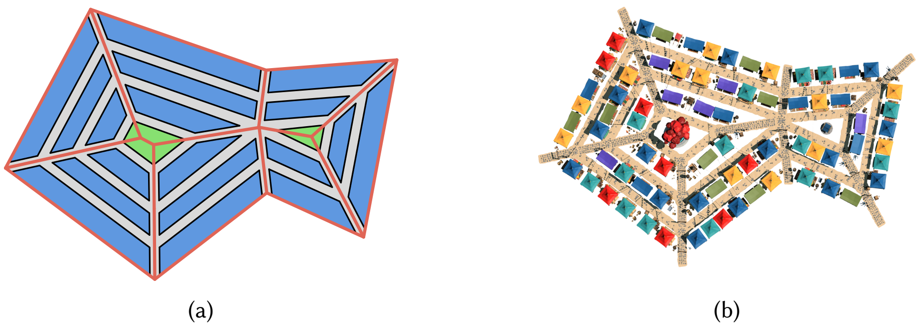


Fig. 3. Market Layout. (a) The straight skeleton of the input polygon (red) defines the market layout. While shrinking the polygon, rings of booth islands (blue) are placed. When no more ring fits, the recursion terminates with a market center (green). The areas are separated with paths (gray). (b) From the straight skeleton layout, we generate the final market with our generation system. For demonstration purposes, the figure does not use random shrinking values.

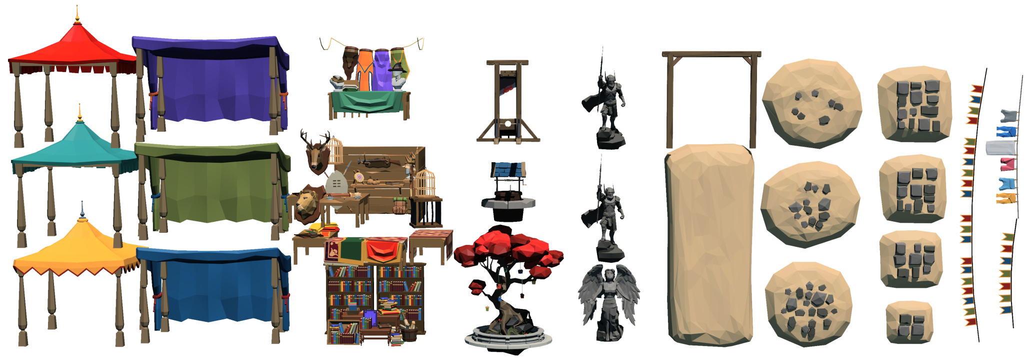


Fig. 4. Market Assets. We show a subset of the assets used to generate the marketplace, including the assets for (left to right) the booth tents, booth fillings, center, path segments, and garlands.

- *The edge event* occurs when an edge of the polygon degenerates, thus when two points merge into one. After the merge, the shrinking process continues with the $n-1$-gon until the three points of a triangle merge into one.
- *The split event* occurs when a concave vertex intersects with an opposing edge, thus when the polygon splits into two. After the split, the two polygons are handled independently from each other. This is easy to implement, since GPU work graphs support tail-recursion.

Let $\bar{t}$ be the distance the polygon can shrink before the next event occurs. Further, we require a user-defined path width w and a minimum and maximum depth d_{min} and d_{max} of a booth island.

When $\bar{t} \geq (w + d_{\text{min}})$, we place a ring by launching a Path node per polygon point along the shrinking trajectory. In between the paths, we launch a BoothIsland node per edge of the polygon. BoothIsland fills the input area with booth assets. This area is always a quadrilateral, passed as a record. We compute the quadrilateral from the polygon's original edge and its shrunk counterpart. To make the layout appear more chaotic, we randomly vary the amount of shrinkage per point and ring between $w + d_{\text{min}}$ and $\min(\bar{t}, w + d_{\text{max}})$. Finally, we recursively launch the market node with the now shrunk polygon.

Otherwise, when $\bar{t} < (w + d_{\text{min}})$, thus when one of the two events is too imminent for another ring to fit, we handle the event instead. Here, we found that simply shrinking the polygon by $\bar{t}$ leaves too much empty space. Instead, we handle the event in a way that fills more of the market area, without exiting the bounds of the original polygon as shown in Fig. 5.

Market Group Implementation. We launch the market node in broadcasting mode with a fixed grid size of $(1, 1, 1)$, thus a single thread group. A thread i of the group is responsible for the point i of the polygon. First, we check for the termination condition of the market recursion, which is given when the area A of the input polygon falls below a threshold and is thus too small for a ring. As the input polygon usually has less than 32 points, and a SIMD lane on a GPU is typically 32 threads wide, we apply wave intrinsics for the area computation using the shoelace formula:

$$A \leftarrow \frac{1}{2}\text{WaveActiveSum}(x_i \cdot y_{i+1} - x_{i+1} \cdot y_i)\,,$$

where x_i, y_i are polygon points, and $i+1$ wraps around the polygon if at the last point.

Next, when A is still sufficiently large, the thread checks at what shrinking value t_i the next event occurs for this point. For this, each thread checks for a merge event with the counter-clockwise adjacent point. If the point is concave, the thread additionally checks for a split event with all

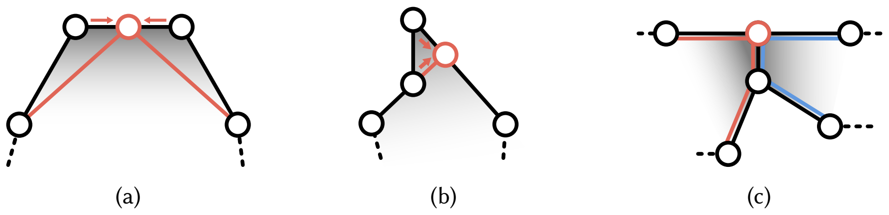


Fig. 5. Market Event Handling. For the edge event, we distinguish between two cases: (a) If both points that are about to merge are convex, we set the position of the merged point to the mean between the two. (b) If one of the points is concave, we project it onto the edge leading to the convex point, clipping away the ear of the polygon. (c) For the split event, we separate the new two polygons with a new edge. The edge starts at the concave point with the event and ends at the closest position from the point to edge the point collides with in the event.

opposing edges. Threads then synchronize to determine the earliest event for the whole polygon, denoted by $\bar{t} = \min(t_0, t_1, \ldots)$. Again, we can use a wave intrinsic:

$$\bar{t} \leftarrow \text{WaveActiveMin}(t_i)\ .$$

Multiple events can occur at the same $\bar{t}$, e.g., for polygons with symmetry. Therefore, we prioritize the event from the lowest thread by finding the minimum $\bar{i}$ of all i that have an event at $\bar{t}$ using wave intrinsics. Finally, the threads cooperatively write the required output records.

Paths. The control parameters for the $\boxed{\text{Path}}$ node consists of the start- and end-point, a width, a random seed, and a quality tier. The tier defines the kind of assets placed along the pathway, e.g., sandy trail or a paved road. As shown in Fig. 3b, the paths along the rings have a lower tier than the paths leading to the market center. To handle path intersections and integrate paths placed by the market with existing paths, we introduce a two phase system: For the first phase, the path node only places an invisible rectangular ray-tracing marker along its trajectory. For the second phase, the path node checks for other paths on its trajectory, using GPU ray tracing. For the intersections shown in Fig. 3b, we place a special asset to improve the look of the intersection. To avoid double placement, only one of the intersecting $\boxed{\text{Path}}$ nodes handles the intersection, e.g., the one with the higher quality tier and lower instance index. With this concept, more complex intersection scenarios can be solved, such as bridges over other paths, or smoothing out paths with splines. This is out of scope of our market example and part of future work. As previously stated, for simplicity reasons, we perform both phases at once by using the slightly outdated BVH of the previous frame.

Garlands. For additional detail, we place garlands across the market booths, as shown in Fig. 1b. Again, we employ a two phase system. For the first phase, the $\boxed{\text{GarlandAnchor}}$ node places markers to indicate its location to others. In the second phase, the node shoots multiple rays into its vicinity to find other anchors. The number of rays is based on the anchor object size so that they cannot miss an adjacent anchor. When a suitable adjacent anchor is found, an additional ray is launched between the two anchor points. The ray assures, no geometry collides with the garland. Then, the $\boxed{\text{GarlandAnchor}}$ node with the lower instance index spawns the garland assets. Ray-tracing markers from different nodes can also interact, e.g., a garland could turn into a clothesline when the ray-tracing finds a path beneath that is of a low quality tier.

7 GROUND CLUTTER

We utilize our system to generate ground clutter in the camera view frustum. This example shows procedural mesh shaders, world grid-based frustum culling, and clutter type classification with ray

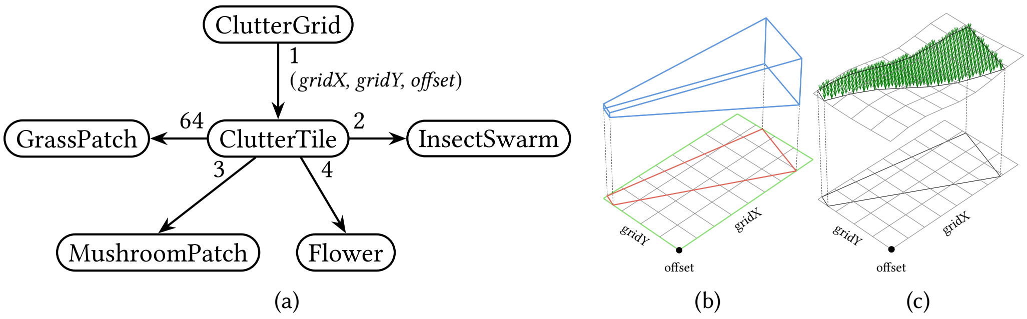


Fig. 6. Clutter Generation Culling. (a) A work graph creates clutter entirely on the GPU. (b) The eight corners of the view frustum (blue) are projected onto the terrain grid plane (red). The required grid of tile nodes to be launched is the intersection of the quantized world-axis-aligned bounding box of the projected corners and the object bounds receiving clutter (green). (c) The individual threads then perform fine grained frustum culling using the height map information.

tracing. The ClutterGrid node of Fig. 6a launches a grid of thread groups of the ClutterTile node, which places different kinds of ground clutter. ClutterGrid is dispatched for the whole scene, or even from other nodes for fine-grained clutter generation in case of multiple isles, for example. The ClutterGrid node issues a ClutterTile node for all terrain tiles that are potentially within the view frustum of Fig. 6b. This is done using a broadcasting launch for ClutterTile. We keep the grid size dynamic by passing it as record together with an offset for one corner of the grid.

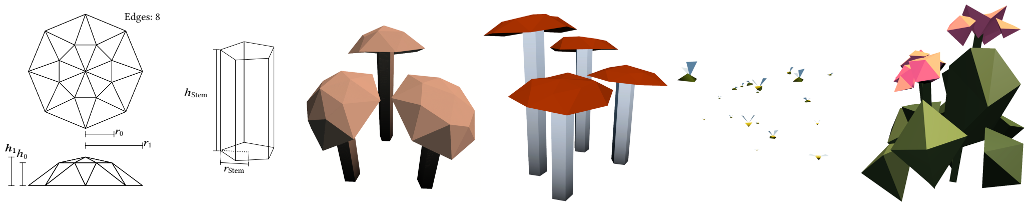
A ClutterTile group consists of an $(8, 8, 1)$ grid of threads, where each thread is responsible for their sub-tile. First, to break up with the regular grid, each thread adds a random offset to its sub-tile position. Then each thread finds the height to its position by sampling the respective height map, and a terrain type map specifying whether to place clutter. Next, fine grained culling is done by each thread individually, as shown in Fig. 6c. Finally, the individual threads, or the entire thread group, decide which of four assets to place as ground clutter.

In case a sub-tile requires *grass* clutter, we launch a GrassPatch mesh shader node. We use the procedural grass mesh shader by Faber et al. [2024]. To avoid grass poking through other low assets, we adjust the height of the grass patch based on the free space upwards. For this, each thread placing a grass patch shoots a ray up and chooses the grass patch height accordingly.

Mushrooms are placed depending on how much sun hits the terrain tile on average. To find this average, all threads of a group trace rays into their positions vicinity: one for where the sun is at 8 am, one for noon, and one for 4 pm. The result of these light rays are then parallely add-reduced in the thread group. If the thread group is dark enough, up to three threads of the group spawn a mushroom patch instead of a grass patch. Similar to the grass patch, a mesh shader uses the parameters of Fig. 7a to create the mushrooms in Fig. 7b.

Flower assets are placed conversely to mushroom patches in light places. They are instanced from a list of predefined flower assets.

Insects are placed depending on the other conditional placements: Near a generated flower, there is a chance to additionally spawn a swarm of bees. When the whole thread group decides to launch grass patch shaders, there is a chance to spawn a swarm of butterflies. As can be seen in Fig. 7c, one mesh shader group generates and renders an insect swarm. To make the insects move convincingly, we add temporal Perlin Noise [Perlin 2002] to their positions. The mesh shader does not receive or load any other external data, except the swarm's location and a time value.



(a) mushroom parameters (b) final mushroom patches (c) bees (d) flower

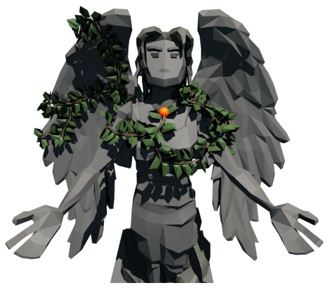
Fig. 7. Stylized Procedural Patches. Parameters for the cap and stem (a) describe the geometry of the mushrooms. In (b) and (c), one mesh shader group creates one type of clutter patch.

8 RESULTS AND DISCUSSION

On our AMD Radeon RX 7900 XTX test system, we use GPU timers to isolate timing measurements for generation and rendering. Note that most of our examples on their own do not fully occupy a GPU, especially because they generate more parallelizable work after running for some time. This will further speed our system once graphic leaf nodes become available. Then, generation algorithms can overlap rendering. To evaluate the performance and adaptability of the ivy generation, we consider different scenarios. Fig. 8 shows that even for an ivy area that covers the majority of the preexisting geometry, the generation is fast enough for real-time editing. Furthermore, the figure demonstrates that the generation adapts to different choices of assets: Figs. 8a - 8c show

(a) 216 instances, 0.51 ms

(b) 216 instances, 0.47 ms


(c) 284 instances, 0.75 ms

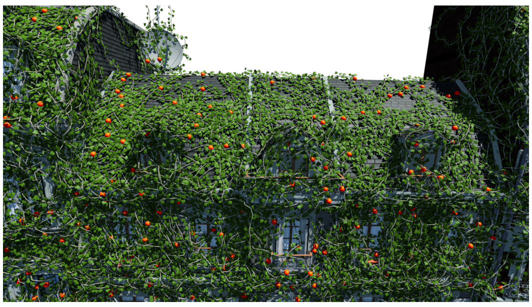
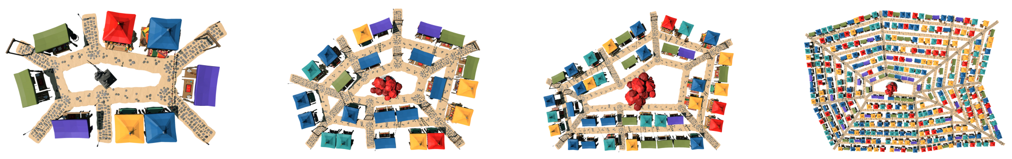
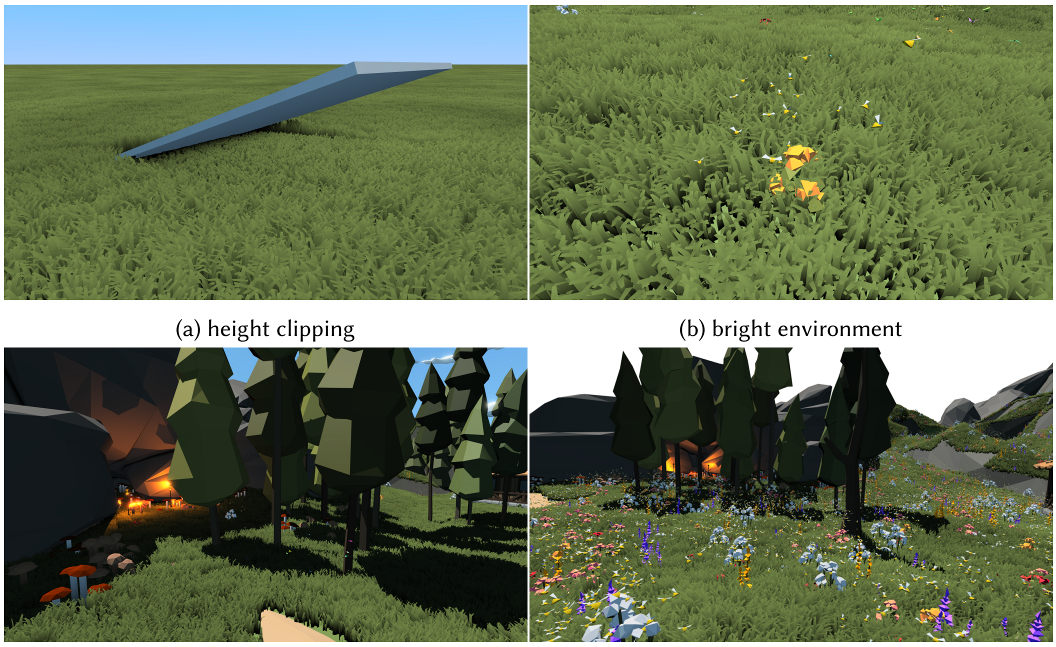

(d) 75,058 instances, 4.17 ms

Fig. 8. Ivy Evaluation. From a start point (red spheres), a single branch adapts to (a) no surface to hang on to, (b) a wall, or (c) a complex surface. (d) A large amount of high-poly branches grow on a detailed surface. Instances count each leaf and branch object separately; timings are generation only.



(a) 525 instances, 0.20 ms (b) 1,230 instances, 0.21 ms (c) 1,624 instances, 0.23 ms (d) 10,145 instances, 0.43 ms

Fig. 9. Market Evaluation for Different Polygons. We measure the number of generated object instances, as well as the generation time.



(a) height clipping (b) bright environment

(c) dark environment (d) high density

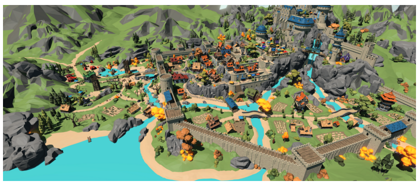
Fig. 10. Clutter Evaluation for Different Scenarios. In (a) some grass patches are obstructed by an object, causing shorter grass. In (b) and (c) different lighting conditions change the kind of clutter. For (d) our system places 79,674 grass patches, 3,359 flowers, 1,038 bee swarms, 3 butterfly swarms, and 281 mushroom patches. The total generation of the draw list takes 0.39 ms.

stylized low-poly assets. In contrast, Fig. 8d uses high-poly ivy assets from Intel Sponza [Meinl et al. 2022] growing on a complex building from Amazon Lumberyard Bistro [Lumberyard 2017]. For comparison, we implemented a feature-equivalent ivy generation with the still experimental Unreal Engine PCG system [Epic Games 2024], where generating a similar amount of ivy as Fig 8d takes over 30 seconds.

Similar to the ivy measurements, we evaluate the market generation for different input polygon configurations. Fig. 9 starts with a small market and increases the polygon size, as well as the number of polygon edges. A plausible layout is generated for all inputs. Even for our largest configuration, where we place over ten-thousand instances of various objects, the generation time is well below what is required for real-time rendering.

View	Mode	Generation	Render	Draw Calls
Overview 79,710 / 79,710	Baseline	3.24 ms	27.74 ms	79,710
	Coalescing	3.14 ms	0.62 ms	965
Market 20,859 / 24,068	Baseline	2.34 ms	8.06 ms	20,859
	Coalescing	2.29 ms	0.51 ms	493

(a)



(b)

Fig. 11. Coalescing Benchmarks. We measure the time to generate and render the augmentation of our test scene. To evaluate our coalescing-based instancing, we measure with and without it. To demonstrate frustum culling, we render the scene from two perspectives: Nothing can be culled for the overview as shown in (b), neither for the generation, nor for rendering. Market, see Fig. 1, where the generation of 55,642 objects can be omitted. Another 3,209 generated objects are culled before rendering.

For the ground clutter in Fig. 10, timings highly depend on the camera placement. Fig. 10a shows the system adjusting the grass patch height based on the available height. Figs. 10b and 10c show how light rays influence the generation by spawning mushrooms where it is dark, and flowers in light areas. Fig. 10d shows that placement with high density is fast enough for real-time rendering.

To evaluate the performance of the coalescing-based instancing, we choose to procedurally augment a scene with all presented examples. Fig. 11 shows the results. As can be seen, using a draw-merging coalescing node greatly reduces the number of draw calls and thus the rendering time. Furthermore, as expected, omitting non-visible generation speeds up the generation stage.

Limitations. Creating geometry directly on the GPU might bear disadvantages: CPU-simulations, like collision detection, must happen on the GPU or the generated data must be transferred. Further, as draw-node scheduling is not deterministic, the element order differs every frame for both the generated draw list and launched mesh nodes. This may cause artifacts when two triangles overlap and makes frame times less predictable. Currently, the maximum work-graph depth is 32, which we occasionally reach. For example, we had to reduce the maximum number of market iterations. Further, there are only eight BVH instance flag bits. This limits our system to eight generation phases. We suggest to use multiple BVHs for different tasks, but this may degrade performance. Finally, geometry generated in a mesh shader is not part of any BVH. Thus, our ground clutter does not use ray-tracing effects.

9 CONCLUSION AND FUTURE WORK

We presented the first system for real-time procedural geometry generation using GPU work graphs. Our approach directly maps generation algorithms into a node-graph hierarchy compatible with work graphs. By utilizing multiple generation phases and GPU ray tracing, we achieve dependent generation. We demonstrated and evaluated our system on three generation examples.

While this work started exploring the possibilities for work-graph-based real-time generation on GPUs, more work is needed to fully realize the potential. We anticipate the emergence of work-graph-based grammars, new work-graph algorithms for planets, terrain, city-layout, building, and road generation, and recursive higher-order surface evaluations. Once graphics leaf nodes become available, we want to explore how to build a BLAS from the output.

ACKNOWLEDGMENTS

We thank Niels Fröhling, Pirmin Pfeifer, Marc Stamminger and the reviewers.

REFERENCES

Oswin Aichholzer, Franz Aurenhammer, David Alberts, and Bernd Gärtner. 1996. *A novel type of skeleton for polygons.* Springer.

Timo Aila and Samuli Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the conference on high performance graphics 2009*. 145–149.

Gill Barequet and Evgeny Yakersberg. 2003. Morphing between shapes by using their straight skeletons. In *Proceedings of the nineteenth annual symposium on Computational geometry*. 378–379.

Blender Online Community. 2024. *Blender 4.1 - a 3D modelling and rendering package.* http://www.blender.org

Kévin Boulanger, Sumanta N. Pattanaik, and Kadi Bouatouch. 2009. Rendering Grass in Real Time with Dynamic Lighting. 29, 1 (2009), 32–41.

Claus Brenner. 2000. Towards fully automatic generation of city models. *International Archives of Photogrammetry and Remote Sensing* 33, B3/1; PART 3 (2000), 84–92.

Cyprien Buron, Jean-Eudes Marvie, Gaël Guennebaud, and Xavier Granier. 2015. Dynamic on-mesh procedural generation. In *Proceedings of Graphics Interface*. Canadian Human-Computer Communications Society, 17–24.

Mark Dokter, Jozef Hladky, Mathias Parger, Dieter Schmalstieg, Hans-Peter Seidel, and Markus Steinberger. 2019. Hierarchical Rasterization of Curved Primitives for Vector Graphics Rendering on the GPU. *Computer Graphics Forum* 38, 2 (2019), 93–103.

Epic Games. 2024. Procedural Content Generation Framework. (2024). https://dev.epicgames.com/documentation/en-us/unreal-engine/procedural-content-generation--framework-in-unreal-engine

Carsten Faber, Max Oberberger, Bastian Kuth, and Quirin Meyer. 2024. Procedural grass rendering. *GPUOpen, March* (2024). https://gpuopen.com/learn/mesh%5Fshaders/mesh%5Fshaders-procedural%5Fgrass%5Frendering/

Zengzhi Fan, Hongwei Li, Karl Hillesland, and Bin Sheng. 2015. Simulation and rendering for millions of grass blades. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games* (San Francisco, California) *(i3D '15)*. ACM, New York, NY, USA, 55–60.

Benedict R. Gaster and Lee Howes. 2012. Can GPGPU Programming Be Liberated from the Data-Parallel Bottleneck? *Computer* 45, 8 (2012), 42–52.

Tobias Hector, Matthäus Chajdas, Maciej Jesionowski, Robert Martin, Qun Lin, Rex Xu, Dominik Witczak, Karthik Srinivasan, Nicolai Haehnle, and Stuart Smith. 2023. *VK_AMDX_shader_enqueue*.

Klemens Jahrmann and Michael Wimmer. 2013. Interactive Grass Rendering Using Real-Time Tessellation. In *WSCG 2013 Full Paper Proceedings* (Plzen, CZ), Manuel Oliveira and Vaclav Skala (Eds.). 114–122.

Kavosh Jazar and Paul G. Kry. 2023. Temporal Set Inversion for Animated Implicits. *ACM Trans. Graph.* 42, 4, Article 134 (jul 2023), 18 pages.

Michael Kenzel, Stefan Lemme, Richard Membarth, Matthias Kurtenacker, Hugo Devillers, Markus Steinberger, and Philipp Slusallek. 2023. AnyQ: An Evaluation Framework for Massively-Parallel Queue Algorithms. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 736–745.

Bernhard Kerbl, Michael Kenzel, Joerg H. Mueller, Dieter Schmalstieg, and Markus Steinberger. 2018. The Broker Queue: A Fast, Linearizable FIFO Queue for Fine-Granular Work Distribution on the GPU. In *Proceedings of the 2018 International Conference on Supercomputing* (Beijing, China) *(ICS '18)*. ACM, New York, NY, USA, 76–85.

Bernhard Kerbl, Michael Kenzel, Dieter Schmalstieg, Hans-Peter Seidel, and Markus Steinberger. 2017. Hierarchical Bucket Queuing for Fine-Grained Priority Scheduling on the GPU. *Computer Graphics Forum* 36, 8 (2017), 232–246.

Bernhard Kerbl, Michael Kenzel, Martin Winter, and Markus Steinberger. 2022. CUDA and Applications to Task-based Programming. Full-day tutorial, presented at Eurographics '22 (Reims, France).

Markus Lipp, Peter Wonka, and Michael Wimmer. 2010. Parallel generation of multiple L-systems. *Computers & Graphics* 34, 5 (2010), 585–593. CAD/GRAPHICS 2009 Extended papers from the 2009 Sketch-Based Interfaces and Modeling Conference Vision, Modeling & Visualization.

Amazon Lumberyard. 2017. Amazon Lumberyard Bistro, Open Research Content Archive (ORCA). http://developer.nvidia.com/orca/amazon-lumberyard-bistro

Jean-Eudes Marvie, Cyprien Buron, Pascal Gautron, Patrice Hirtzlin, and Gaël Sourimant. 2012. GPU Shape Grammars. *Computer Graphics Forum* (2012).

Frank Meinl, Katica Putica, Cristiano Siqueria, Timothy Heath, Justin Prazen, Sebastian Herholz, Bruce Cherniak, and Anton Kaplanyan. 2022. Intel Sample Library. https://www.intel.com/content/www/us/en/developer/topic-technology/graphics-processing-research/samples.html

Paul Merrell, Eric Schkufza, and Vladlen Koltun. 2010. Computer-generated residential building layouts. In *ACM SIGGRAPH Asia 2010 papers*. 1–12.

Microsoft Cooperation 2024. *DirectX-Specs.* Microsoft Cooperation. https://github.com/microsoft/DirectX-Specs

Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. 2006. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers*. 614–623.

NVIDIA 2024. *CUDA C++ Programming Guide*. NVIDIA.

Timo Oksanen and Arto Visala. 2009. Coverage path planning algorithms for agricultural field machines. *Journal of field robotics* 26, 8 (2009), 651–668.

Marc S. Orr, Bradford M. Beckmann, Steven K. Reinhardt, and David A. Wood. 2014. Fine-grain task aggregation and coordination on GPUs. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 181–192.

Dimitris Papavasiliou. 2015. Real-time grass (and other procedural objects) on terrain. *Journal of Computer Graphics Techniques (JCGT)* 4, 1 (2015), 26–49.

Yoav IH Parish and Pascal Müller. 2001. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 301–308.

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: a general purpose ray tracing engine. *ACM Trans. Graph.* 29, 4, Article 66 (jul 2010).

Anjul Patney, Stanley Tzeng, Kerry A. Seitz, and John D. Owens. 2015. Piko: a framework for authoring programmable graphics pipelines. *ACM Trans. Graph.* 34, 4, Article 147 (jul 2015), 13 pages.

Ken Perlin. 2002. Improving noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. 681–682.

P. Prusinkiewicz and Aristid Lindenmayer. 1990. *The algorithmic beauty of plants*. Springer-Verlag, Berlin, Heidelberg.

Thomas R.W. Scogland and Wu-chun Feng. 2015. Design and Evaluation of Scalable Concurrent Queues for Many-Core Architectures. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (Austin, Texas, USA) *(ICPE '15)*. ACM, New York, NY, USA, 63–74.

Side Effects Software Inc. 2023. *Houdini 20*. https://www.sidefx.com/products/houdini/

Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel, and Dieter Schmalstieg. 2012. Softshell: dynamic scheduling on GPUs. *ACM Trans. Graph.* 31, 6, Article 161 (nov 2012), 11 pages.

Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. 2014a. Whippletree: task-based scheduling of dynamic workloads on the GPU. *ACM Trans. Graph.* 33, 6, Article 228 (nov 2014), 11 pages.

Markus Steinberger, Michael Kenzel, Bernhard Kainz, Jörg Müller, Peter Wonka, and Dieter Schmalstieg. 2014b. Parallel Generation of Architecture on the GPU. *Computer Graphics Forum* (2014).

George Stiny and James Gips. 1971. Shape Grammars and the Generative Specification of Painting and Sculpture. *IFIP Congress* 71, 1460–1465.

Stanley Tzeng, Anjul Patney, and John D. Owens. 2010. Task management for irregular-parallel workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics* (Saarbrucken, Germany) *(HPG '10)*. Eurographics Association, 29–37.

Carlos A Vanegas, Daniel G Aliaga, Peter Wonka, Pascal Müller, Paul Waddell, and Benjamin Watson. 2010. Modelling the appearance and behaviour of urban spaces. In *Computer Graphics Forum*, Vol. 29. Wiley Online Library, 25–42.

Carlos A Vanegas, Tom Kelly, Basil Weber, Jan Halatsch, Daniel G Aliaga, and Pascal Müller. 2012. Procedural generation of parcels in urban modeling. In *Computer graphics forum*, Vol. 31. Wiley Online Library, 681–690.

Martin Winter, Mathias Parger, Daniel Mlakar, and Markus Steinberger. 2021. Are dynamic memory managers on GPUs slow? A survey and benchmarks. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 219–233.