Real-Time Layered Materials Compositing Using Spatial Clustering Encoding

Sergey Makeev

1.1 Introduction

Most of the modern rendering engines take advantage of using a library of simple and well-known materials and a layered material representation to author detailed and high-quality ingame materials. Popular tools used in texturing pipeline nowadays (e.g. "Allegorithmic Substance Painter" and "Quixel NDO Painter") are also based on the concept of layered materials [Neubelt and Pettineo 13, Deguy et al. 16, Karis 13].

In this chapter, we present an algorithm that uses a layered materials method which allows us to create composite materials using a large number of layers in real-time. Our algorithm is designed to mimic "Allegorithmic Substance" texture pipeline as close as possible but in real-time. The proposed technique based on the blending of multiple well-known materials where a shared materials library defines the surface properties for each material used in compositing.

Using our method, each mesh can have one unique UV-set and several unique texture blend masks where each blend mask defines the per-pixel blending weights for the material from the library. Each material from the library can use a detail textures technique to improve surface details resolution. Using the materials with detail textures for the composition has the advantage of breaking the texture resolution barrier and allows us to produce a final composition at a very high resolution. Having high-resolution in-game materials is especially crucial in the 4K era.

Our method supports the replacement of a library of materials and transparency modifications for the texture blend masks at runtime. Material replacement at runtime leads to a different visual appearance of the resulting composited material which is especially important for games supporting User Generated Content or in-game Customization. The presented technique is used for rendering armored vehicles in "Armored Warfare", an action multiplayer tank game published by My.com.

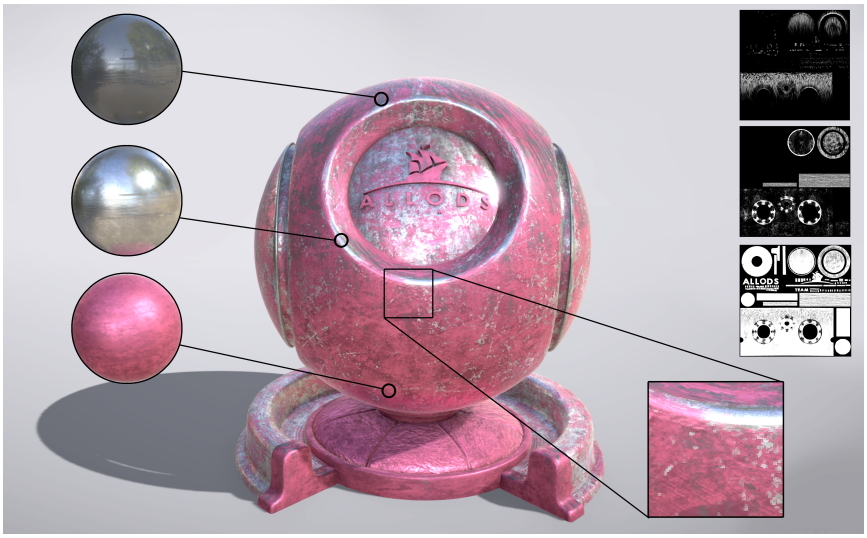

Figure 1.1. An example of Dynamic Material Layering. This example mesh uses three texture blend masks to define the blending transparency and three library materials to define the surface properties.

1.2 Overview of Current Techniques

A popular method for compositing is to pre-bake a multilayered material into a set of textures (albedo, normal, roughness, etc.) that are used by the game engine. The resulting texture set are primarily designed for a specific mesh and cannot be shared between different meshes. We call this method "Static Material Layering." [Neubelt and Pettineo 13,Deguy et al. 16]. This approach gives good results but requires a lot of GPU memory to store the final high-resolution textures.

To break this limitation, some modern rendering engines use a technique that is pretty similar to the method that has already been proven for rendering terrain which is called "Texture Splatting." [Charles Bloom 00]. To produce the final composited texture, these engines blend several textures using a pixel shader and a set of texture blend masks which define the transparency of the blending. We call this method "Dynamic Material Layering." [Inside Unreal 13, Noguer 16]. This approach works well as shown in Figure 1.1 but is limited to a small number of simultaneous material layers due to memory and performance limitations.

1.3 Introduced Terms

Since different game engines and material authoring pipelines use different terms, here are definitions which are used in this article.

- **Material Template.** One single well-known material such as gold, steel, wood, etc. Material Template can use a tiled detail texture to give the illusion of greater detail for a material. Material Templates are used as basic blocks to create complex multi-layered materials.
- **Material Mask.** A grayscale texture which is used for defining transparency while compositing different Material Templates. Usually these textures are created by modern texturing tools like "Allegorithmic Substance Painter" and "Quixel NDO Painter" using a semi-procedural approach.
- **Color ID.** A color-coded texture which defines areas of UVs that belong to different opaque materials. The opaque material does not have a blend mask associated with it, and it is always used as a bottom layer in our composition. A color-coded representation where each unique color represents a single material is used to simplify the content pipeline and reduce the number of required textures. Each color-coded texture can represent several opaque materials as shown in Figure 1.2.
- **Layered Material.** This is a material definition which is used to build the final composite material. Each Layered Material has a single Color ID associated with it and an ordered set of Material Masks which define the composition order and the blending weights of the materials. Layered Material also has a set of Material Templates associated with it to define the visual appearance of each material used in a composition.

1.4 Algorithm Overview

Existing solutions [Inside Unreal 13, Noguer 16] which used for Dynamic Material Layering store transparency for different materials in RGBA channels of the texture. When each material mask covers only a small area of the texture such solutions are inefficient in terms of memory consumption. A lot of texture space is not used for the composition and wasted.

We observe that blend masks are usually coherent in the texture space and only partially overlap each other. Using this observation, we propose

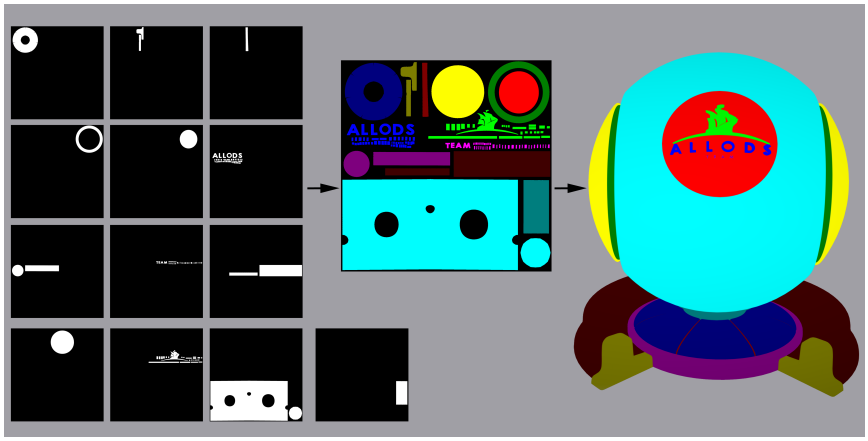

Figure 1.2. Several opaque material masks combined into a single Color ID map and applied to the mesh.

storing the different non-overlapped blend masks in the same texture channels. To achieve this, we group several Material Masks into a set that we call Clusters. We build the clusters based on the connectivity between texels in texture space. This allows us to use the texture space more effectively since the different clusters can store their blend masks in the same shared texture channels. At runtime we use this clustered representation and the set of the material templates to make the final material composition as shown in Figure 1.3.

Since we are storing the different blend masks in the same texture channels, it is critical to take into account texture filtering boundaries between different clusters. Texture filtering of different blend masks lead to errors during the composition stage due to the leaking of texture blend masks from one material into another. While building the material clusters, we consider which neighboring pixels are involved in the texture filtering and this information is used while creating the clusters.

To create the initial partitioning into the clusters, we perform a connectivity analysis for the set of Material Masks. Connectivity analysis classifies all the texels which are used for texture filtering as connected. If the texels are classified as connected, they will belong to the same material cluster. When we perform a connectivity analysis, we should also take mipmap texture filtering into account. At the same time, we should limit the number of supported mipmap levels otherwise at the very last mipmap level all the texels will be classified as connected. For our implementation, we decided to support only the first four mipmap levels. Smaller mipmap levels are not handled by our implementation and discarded. Supporting

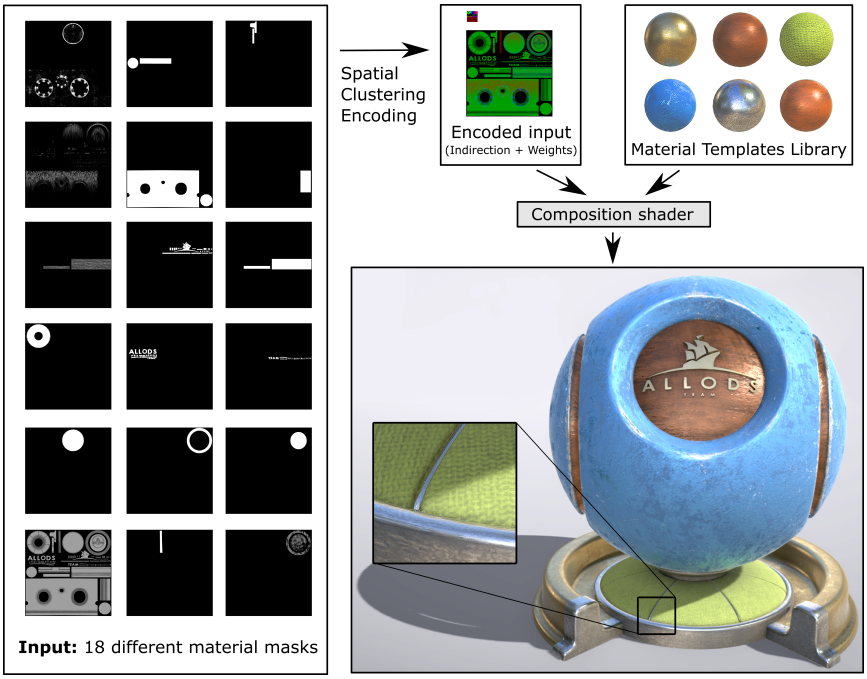

Figure 1.3. An example of the use of the presented technique. Several texture blend masks encoded as a single RGB weights texture and a single cluster indirection texture. Encoded material blend masks and material templates from the library are composited to get the final image.

only the first four mipmap levels is enough to preserve a good quality of the texture filtering and keep the number of the resulting clusters small. An incomplete mipmaps chain might lead to aliasing, but its level is acceptable [Mittring 08]. In practice, the resulting aliasing can be barely visible and effectively removed by most of the modern anti-aliasing algorithms.

Each resulting cluster should not contain more than a limited number of materials where the number of materials depends on how many per pixel material layers we need to support. In practice, the number of materials used in the cluster is usually equal to four or five since we store the cluster blend weights in the BC1 or BC3 texture format. A practically unlimited total number of materials and five per-pixel materials is enough to represent even a very complex layered material.

Constructing the clusters with a limited number of materials is not always possible since we can find more connected materials than the maximum allowed number of materials per cluster. As a result, we can find a

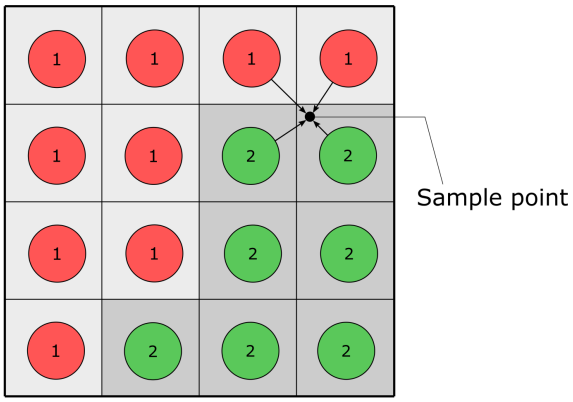

Figure 1.4. The texture unit uses blend masks from different Clusters during texture filtering. The result of such filtering is a leaking of the material boundaries which leads to visual artifacts in the composition stage.

cluster which is used more than a maximum allowed number of materials. We can split such clusters into several smaller ones that meet our initial requirements. This will lead to a texture filtering error for the texels shared between the clusters edges. Filtering errors will occur due to erroneous texture filtering between blend masks from the different clusters in which different materials are encoded (see Figure 1.4). We propose a solution that minimizes leaking of the texture blend masks while splitting such clusters. For more details see Section 1.5.7.

1.4.1 Spatial Clustering Encoding Representation

At the preprocessing stage, we build a clustered representation using a single Color ID to define all the opaque materials and an ordered set of Material Masks to define all transparent materials. As a result of preprocessing, we get a dataset which is used for the runtime composition and contains three different types of data.

- **Cluster Indirection**. An indirection texture that defines the current cluster ID for a specified texel. The cluster ID is stored using an integer texture format and defines which set of materials should be used for a given texel. Cluster Indirection is stored using a texture that has a lower resolution than a resolution of the source Material Masks. Neighbor texels usually use the same set of materials and the set of used materials rarely changes which allows us to use a smaller resolution texture to store this data. Since this texture contains the

integer data, this texture cannot use any texture filtering. Cluster Indirection is stored in the texture without mipmaps and fetched using a POINT texture filtering mode.

- **Cluster Weights**. The weight texture defines the blending weight for each material in a set of materials which a specified by the cluster ID. We support up to five different material masks per pixel where the weights are stored using the BC3 texture format. Cluster Weights are stored using a texture that has the same resolution, as the input Material Masks, despite the set of used materials rarely vary, neighbor texels of a blend mask can differ significantly. Cluster Weights can be correctly filtered inside the same material cluster. Texture data is stored with mipmaps and fetched using a TRILINEAR or ANISOTROPIC texture filtering mode.
- **Cluster Properties**. Defines material surface properties such as albedo, roughness, metalness, etc. which are used for the final composition. We decided to use a Structured Buffer to store the Cluster Properties. Depending on the implementation, Cluster Properties can also be stored in the Constant Buffer.

At the composition stage, we obtain the cluster ID for each fragment which defines a set of used materials and the blending weights. Then using the Cluster Properties, we obtain the surface properties for each material which are used in the current cluster. Afterwards, we use the blend weights and the surface properties for the final composition of the surface properties for a given fragment. See Figure 1.5 for more details.

1.4.2 Order-independent representation for blend masks

For the final composition, we need to blend materials in the correct order, as defined in the input data. The most common way of doing this is to perform alpha blending and composite the fragments in a back-to-front order using the following equation:

$$C_{final} = C_{src} * \alpha + C_{dst} * (1 - \alpha)$$

Then we repeat this operation for all the blend masks used for blending:

$$C_{final} = [C_n\alpha_n + (1 - \alpha_n) \dots [C_2\alpha_2 + (1 - \alpha_2) [C_1\alpha_1 + (1 - \alpha_1)C_0]]]$$

This approach depends on the order of operations and instead of using alpha-blending we can rewrite the blending equation in weighted form:

$$C_{final} = w_0C_0 + w_1C_1 + w_2C_2 + \dots + w_nC_n$$

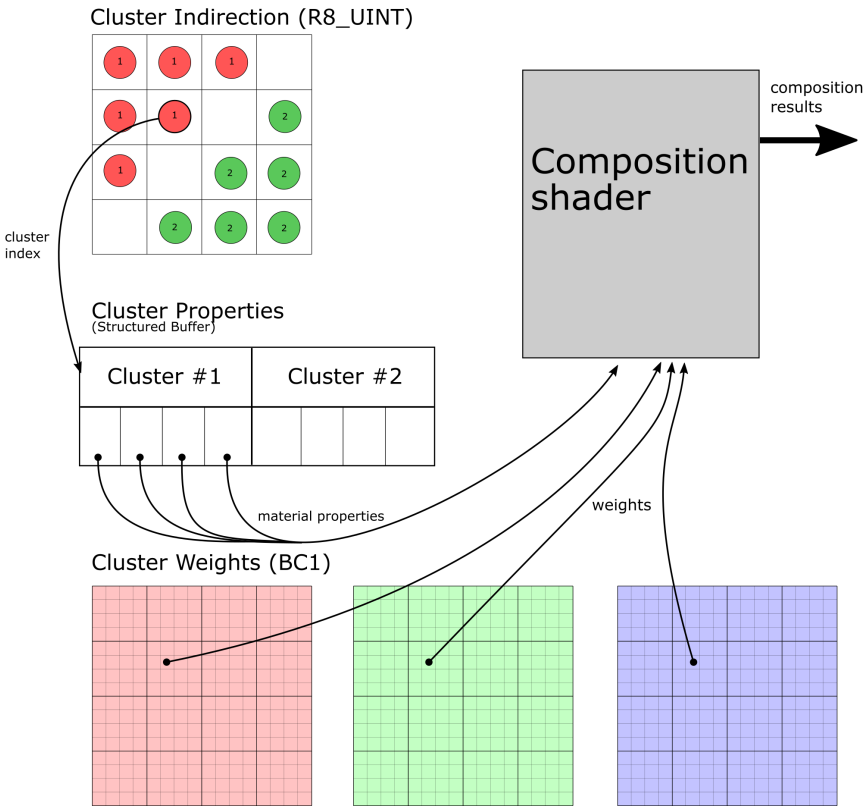

Figure 1.5. An example representation and usage of the encoded data.

Where:

$$w_k = (1 - \alpha_n)(1 - \alpha_{n-1}) \dots (1 - \alpha_{k+1})\alpha_k$$

Since the resulting weights are normalized, we know that the sum of all weights are always equal to one. We can use this property to reconstruct one of the weights inside a composting pixel-shader instead of storing this weight in the texture channel:

$$w_n = 1 - w_0 - w_1 - w_2 \cdots - w_{n-1}$$

For our implementation, we decided to use the order-independent weighted representation for the texture blend masks. The order-independent representation allows us to swap the texture channels inside the cluster freely. This property can be very useful for several further optimizations.

1.5 Algorithm Implementation

1.5.1 Extract background materials

First, we define an opaque material for each texel. The opaque material is also used to determine which texels are inside the UV mapping and which ones are not. Then we determine which texture resolution we should use for the latest supported mipmap level. For each texel inside the mip-level, we generate a Color ID value from the original high-resolution Color ID texture. To avoid situations where the resulting mipmap texel uses several different opaque materials, we forbid using different Colors IDs in the neighboring texels. This natural limitation allows us to find potential clusterization issues at the very early stage of the art-pipeline and helps create Color ID maps which can be efficiently clustered. This also allows us to skip the connectivity analysis for all the opaque materials since the different opaque materials never share the neighboring texels.

1.5.2 Material layers

At this step, we have the ordered set of texture blend masks called Material layers. Each material layer defines the transparency of the blending for each texel. We downsample each material layer using a MAX filter to the resolution corresponding the latest supported mipmap level. If the resulting texel was marked as unused on the previous step, this texel is located outside of the valid UV mapping and will not be used in the composition.

1.5.3 Weighted sum representation

At this step, we have several downsampled texture blend masks defined in the specified order used for the alpha blending. We transform the texture blend masks to a normalized weighted form using Algorithm 1. The normalized weighted representation also helps us to discard texels that are entirely covered by other materials and do not contribute to the final composition.

1.5.4 Undirected graph representation

At this step, we move from a bitmap representation of input data to an undirected graph representation. The advantage of a graph representation in comparison with a bitmap representation is that we can use graph theory for analyzing and building clusters with specific characteristics. For each texel, we find all the texture blend masks that have contributed to a given texel, and assign a unique texel identifier that corresponds to the unique combination of blend masks used. Then we find the connected area using an algorithm similar to a flood-fill algorithm and make a separate

```
for every texel(X,Y) in opaque layer do
    if texel(X,Y) is empty in opaque layer then
        skip texel
    end
    accum = 1.0
    for every input texture blend mask do
        alpha = blend_mask(X,Y)
        layer_weight(X,Y) = alpha * accum
        accum = accum * (1.0 - alpha)
    end
end
```

Algorithm 1: Converting the texture blend mask to a normalized weighted form.

graph vertice from each unique combination. The result produced by the algorithm shown in Figure 1.6. For each resulting graph vertice, we store an assigned identifier that encodes which blend masks were used to build this graph vertice.

1.5.5 Texture filtering requirement analysis

At this step, we build edges between graph vertices according to the following rule: If two graph vertices have adjacent pixels that are used for texture filtering, these vertices are connected.

As a result, we built a connected undirected graph $G = (V,E)$ from the source bitmap data. V represents the area affected by the different combinations of input texture blend masks. E represents the texture filtering relationship between these areas as shown in Figure 1.7. The number of texels used for texture filtering determines the edge weight and will be used later for the graph partitioning.

1.5.6 Finding the number of connected components

The resulting undirected graph usually has several connected components. Our goal is to find the number of connected components and split the graph into several subgraphs between which no filtering is required (see Figure 1.8). Each subgraph is processed as an independent graph for the next algorithm steps. If the resulting graph already fits our initial requirements and does not exceeding the maximum allowed number of materials, then the next step is redundant and can be skipped. Otherwise, the next algorithm step splits the graph into several subgraphs with specific properties defined by our initial requirements.

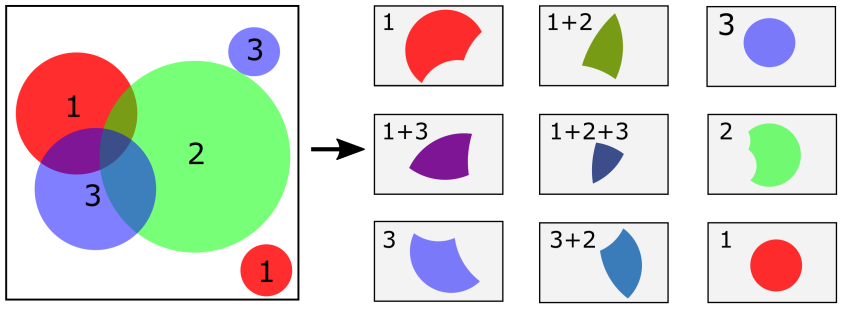

Figure 1.6. Spliting bitmap data to the graph vertices. Red, Green and Blue circles represent areas covered by the different texture blend masks.

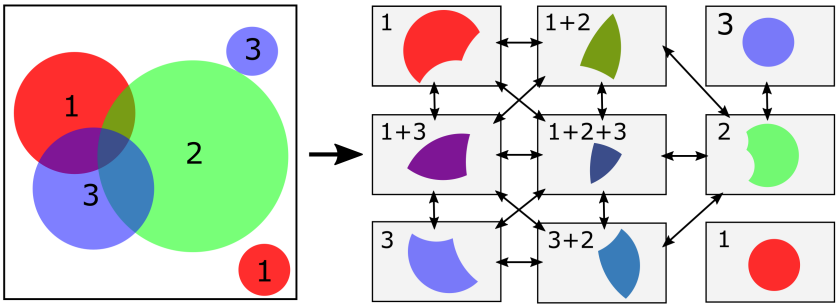

Figure 1.7. Resulting undirected graph. Arrows represent edges which indicate the texture filtering relationships between vertices.

1.5.7 Solving the graph partitioning problem

At this step, we need to solve the graph partitioning problem and split a graph $G = (V,E)$, where V is the set of vertices and E are edges, into smaller components with specific properties. Typically, graph partition problems are NP-hard problems so we should use heuristics and approximations to solve the graph partition problem. To find the optimal solution, we use an iterative greedy algorithm to find a set of edges with minimal weight to cut. Our solution is inspired by the heuristic algorithm of graph partitioning proposed by Kernighan and Lin [Kernighan and Lin 70].

Our goal is to divide the graph into subsets A and B where subset A satisfies initial requirements, and the sum of edge weights from A to B are minimized. Since the weight of the edge is the number of pixels used for the texture filtering, by minimizing the sum of edge weights from A to B we reduce the resulting filtering error. Our multi-pass algorithm maintains

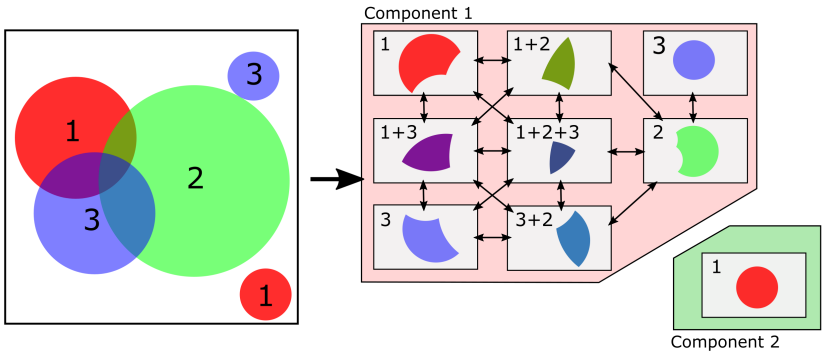

Figure 1.8. A resulting graph with two connected components.

and improves a partition, using a greedy algorithm in each pass to pair up vertices of A with vertices of B, so that moving the paired vertices from one side of the partition to the other improves the partitioning.

Next, our algorithm chooses the best solution from all solutions that have been tried. Thus our algorithm attempts to find the optimal subset A which has a minimum sum of edge weights to cut. See Algorithm 2 for implementation details. To demonstrate one iteration step of the algorithm, see Figure 1.9.

1.5.8 Generating the final data

At the final step, we have a set of undirected graphs, where each graph fits our initial requirements. In practice, many resulting graphs use fewer materials than the maximum allowable number. To reduce the resulting number of clusters, we combine such graphs into larger ones as long as the merged results satisfy the initial requirements.

Next, we a build normalized weighted representation as described in Section 1.5.3, but this time for full-resolution texture data. For each resulting graph, we copy all texels used by the graph vertices from the weight textures into separate channels of the Cluster Weights texture. Then we store the used graph index into a Cluster Indirection texture using the R8_UINT format. Next, for the Cluster Weights, we generate a partial mipmap chain which is limited by the number of supported mipmaps and then compress the resulting texture using the BC1 or BC3 texture format. See Figure 1.10 for the example set of resulting textures.

```
1  foreach source layers identifier existing in the graph G(V,E) do
2  |  begin split graph into initial subsets A and B
3  |  |  add all graph vertices with same identifier to subset A
4  |  |  add all other graph vertices into subset B
5  |  end
6  |  begin_loop
7  |  |  calculate sum of edges weights between subset A and B
   |  |   and store as solution
8  |  |  find a vertice inside a subset B that has the largest sum of
   |  |   edges crossing between subsets and can be moved to a
   |  |   subset A without violating our constraints
9  |  |  if such vertice found then
10 |  |  |  move all vertices with same identifier as a found vertice
   |  |  |   from the subset B to the subset A
11 |  |  else
12 |  |  |  break
13 |  end_loop
14 end
15 return solution with the minimal weight between subsets
```

Algorithm 2: Graph partitioning algorithm.

1.5.9 Runtime Composition

For the final composition of the material at runtime, we use the following approach:

- Fetch the encoded cluster ID from the indirection texture.
- Fetch material blend weights from the weight texture.
- Read the surface properties stored in the Structured Buffer using the fetched cluster ID.
- Make the final composition using the obtained surface parameters and material blend weights. See Listing 1.1 for an example of a basic compositing shader.

Since we use a normalized weighted representation for storing the blend weights, we can change the blend weight of any individual material layer and renormalize the total sum of the weights. This allows us to change the transparency of individual layers at runtime.

Textures used for the composition usually have insufficient resolution. To increase the final resolution of the composition, we use detail textures stored in the texture arrays as described in [Hamilton and Brown 16]. To

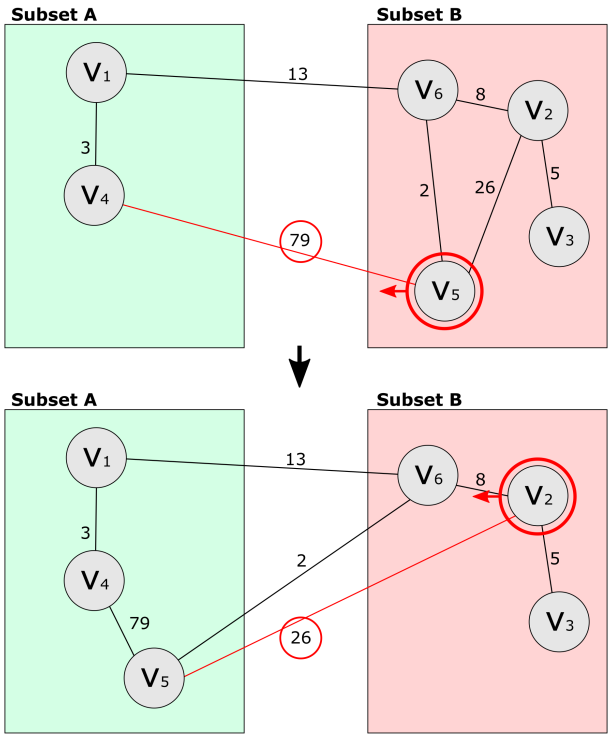

Figure 1.9. One step of the graph partitioning algorithm. The veritice V_5 moved from subset B into subset A.

generate UV-set for detail textures, we multiply original UV-set by tiling factor. Tiling factor for detailed UV-set can be specified per material. We use two sets of texture arrays: First for the surface parameters (albedo, roughness, metallic) and the second for the normal maps. Each material used in the composition can use an arbitrary detail map for the surface properties and an arbitrary detail map for the surface normals. See Figure 1.11 for examples of the composition with and without detail textures.

For the normal maps blending, we use a weighted blending of partial derivatives as described in "Blending in Detail" [Barre-Brisebois and Hill 12]. Additionally, we can blend only two detail maps with the highest contribution weights at a medium distance and completely disable detail maps at a long distance to improve the composition performance.

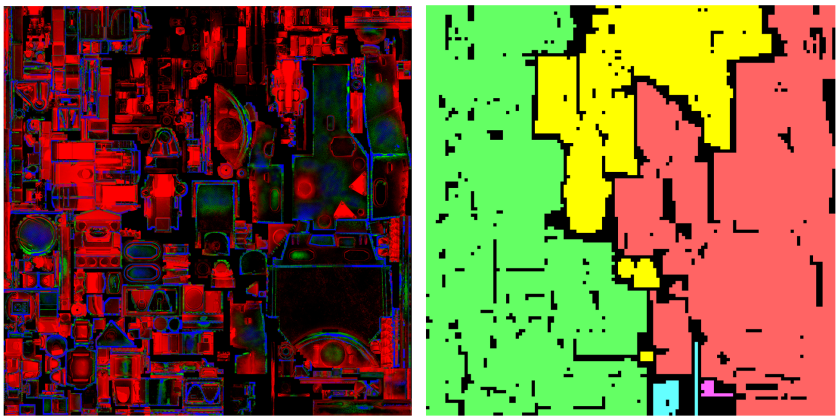

Figure 1.10. Final data generated by our implementation. Cluster weight texture (left) and Indirection texture (right). Indirection texture is colorized and upscaled by 8 times for demonstration purposes. Image courtesy of Mail.Ru Group.

Figure 1.11. Rendering using detail maps (left) and without detail maps (right). Image courtesy of Mail.Ru Group.

1.5.10 Source code

For demonstration purposes, we implemented our method using C# for preprocessing and the Unity game engine by Unity Technologies for the runtime materials composition. Full source code can be found in the sup-

Technique	Per-pixel number of layers	Total number of layers	Per-pixel number of detail-textures	ALU	TEX
Unreal Engine 4 - MatLayerBlendSimple	4	4	4	82	14
Unreal Engine 4 - Custom material	3	3	0	23	2
Spatial Clustering Encoding - High	5	9+	5	140	13
Spatial Clustering Encoding - Standard	4	9+	4	104	11
Spatial Clustering Encoding - Fast	4	9+	1	42	4
Spatial Clustering Encoding - Fastest	3	9+	0	26	3

Table 1.1. Number of instructions resulting for different layered material techniques.

plemental materials of this book.

Source code is also available at https://github.com/SergeyMakeev/GpuZen2

1.6 Results

We used the approach described in this article for rendering the armored vehicles in the "Armored Warfare" game. In our game, users can customize coloring and materials used for rendering the armored vehicles. The presented technique allows minimizing the number of textures stored on disk while supporting high-quality textures and allowing the customization of the visual appearance. You can see some results of using our technique in Figure 1.12. We compared the number of instructions resulting for our technique and number of instructions resulting for Unreal Engine 4 material layering technique, see Table 1.1. Table 1.2 shows build times and the number of resulting materials clusters made for the armored vehicle.

Asset	Input materials count	Build time	Mips count	Graph vertices count	Clusters count	Memory used for cluster parameters
Turret	9	1.9 sec	3	1791	5	560 bytes
Hull	7	1.4 sec	3	3937	3	336 bytes
Cannon	7	0.5 sec	3	262	4	448 bytes
Wheels	6	0.9 sec	3	1422	3	336 bytes
Tracks	5	0.2 sec	3	268	1	112 bytes

Table 1.2. Build time and resulting cluster statistics for example asset.

1.7 Conclusion and Future Work

The method described in this chapter helps to store efficiently and use more texture blend masks than would be allowed by existing methods with some natural limitations. Also, the proposed method supports real-time material recomposition using the proposed data representation. For the most effective use of our method, it is necessary to take into account the texture space connectivity of the different texture blend masks at the earliest stages ofthe art-pipeline. At the same time, the proposed method is suitable for any existing art assets without additional preparation with some tolerable texture filtering errors. We are continue to develop and refine of the proposed technique. Here are some areas for further development:

- Nonlinear blending for the material composition as proposed in [Hardy and Roberts 06].
- Reducing the texture filtering errors when dividing clusters. Since the texture blend masks are order independent, we can swap the texture channels inside the cluster. Using the least squares minimization technique along the "seams" boundary as proposed in [Iwanicki 13], we can reduce the texture filtering error almost to zero.
- Composition after evaluating BRDF instead of the surface properties composition. Using this approach we can create accurate multilayered materials with multiple specular lobes.

- Using the vertex color as a blend weight modifier for local dynamic material recomposition (dynamic dirt, scratches, etc.).

1.8 Acknowledgements

First, I would like to thank Vladimir Egorov, my friend and colleague, for his suggestions and early feedback on this article. Peter Sikachev, Vadim Slyusarev, Bonifacio Costiniano and Alexandre Chekroun for their feedback on this article. In addition, I would like to thank all Allods Team members as well.

```
1  struct SurfaceParameters
2  {
3   float3 albedo;
4  };
5  struct ClusterParameters
6  {
7   SurfaceParameters layer0;
8   SurfaceParameters layer1;
9   SurfaceParameters layer2;
10  SurfaceParameters layer3;
11 };
12 // Weights texture
13 Texture2D cWeights;
14 // Indirection texture
15 Texture2D cIndirection;
16 // Material parameters (stored per cluster)
17 StructuredBuffer<ClusterParameters> clusterParameters;
18
19 float4 DecodeAndComposition( float2 uv ) : SV_Target0
20 {
21    float4 weights;
22    // Fetch weights
23    weights.xyz = cWeights.Sample(samplerTrilinear, uv).rgb;
24
25    // Reconstruct weight
26    weights.w = 1.0 - weights.x - weights.y - weights.z;
27
28    // Fetch index
29    uint clusterIndex = cIndirection.Sample(samplerPoint, uv).r;
30
31    // Get material params
32    ClusterParameters params = clusterParameters[clusterIndex];
33
34    // Use the material parameters and weights
35    //                      for a final composition
36    float3 albedo = params.layer0.albedo * weights.x +
37                    params.layer1.albedo * weights.y +
38                    params.layer2.albedo * weights.z +
39                    params.layer3.albedo * weights.w;
40
41    return float4(albedo, 1.0);
42 }
```

Listing 1.1. An example of cluster decoding.

Figure 1.12. An example of composited material and some material templates used in composition (Top) and the same composited material with different material templates applied (Bottom). Image courtesy of Mail.Ru Group.

Bibliography

[Barre-Brisebois and Hill 12] Colin Barre-Brisebois and Stephen Hill. "Blending in Detail.", 2012. Available online (http://blog.selfshadow.com/publications/blending-in-detail/).

[Charles Bloom 00] Charles Bloom. "Terrain Texture Compositing by Blending in the Frame-Buffer.", 2000. Available online (http://www.cbloom.com/3d/techdocs/splatting.txt).

[Deguy et al. 16] Sebastien Deguy, Rogelio Olguin, and Brad Smith. "Texturing Uncharted 4: a matter of Substance." *GDC2016*.

[Hamilton and Brown 16] Andrew Hamilton and Ken Brown. "Photogrammetry and Star Wars Battlefront." *GDC2016*.

[Hardy and Roberts 06] Alexandre Hardy and Duncan Andrew Keith Mc Roberts. "Blend maps: enhanced terrain texturing." *SAICSIT2006*.

[Inside Unreal 13] Inside Unreal. "A Look at Unreal Engine 4 Layered Materials.", 2013. Available online (https://www.unrealengine.com/news/look-at-unreal-engine-4-layered-materials).

[Iwanicki 13] Michal Iwanicki. "Lighting technology of The Last of Us." *SIGGRAPH2013*.

[Karis 13] Brian Karis. "Real Shading in Unreal Engine 4 : Physically Based Shading in Theory and Practice." *SIGGRAPH2013*.

[Kernighan and Lin 70] Brian W. Kernighan and Shen Lin. "An efficient heuristic procedure for partitioning graphs." *The Bell System Technical Journal. Vol 49.*, pp. 291–307.

[Mittring 08] Martin Mittring. "Advanced Virtual Texture Topics." *SIGGRAPH2008*.

[Neubelt and Pettineo 13] David Neubelt and Matt Pettineo. "Crafting a Next-Gen Material Pipeline for The Order: 1886." *SIGGRAPH2013*.

[Noguer 16] Jeremie Noguer. "The Next Frontier of Texturing Workflows.", 2016. Available online (https://www.allegorithmic.com/blog/next-frontier-texturing-workflows).