Designing Gratin, A GPU-Tailored Node-Based System

Romain Vergne, Pascal Barla

Designing Gratin, A GPU-Tailored Node-Based System

Romain Vergne[1,3] Pascal Barla[2,3]
[1]Univ. Grenoble Alpes [2]Univ. Bordeaux [3]CNRS; Inria

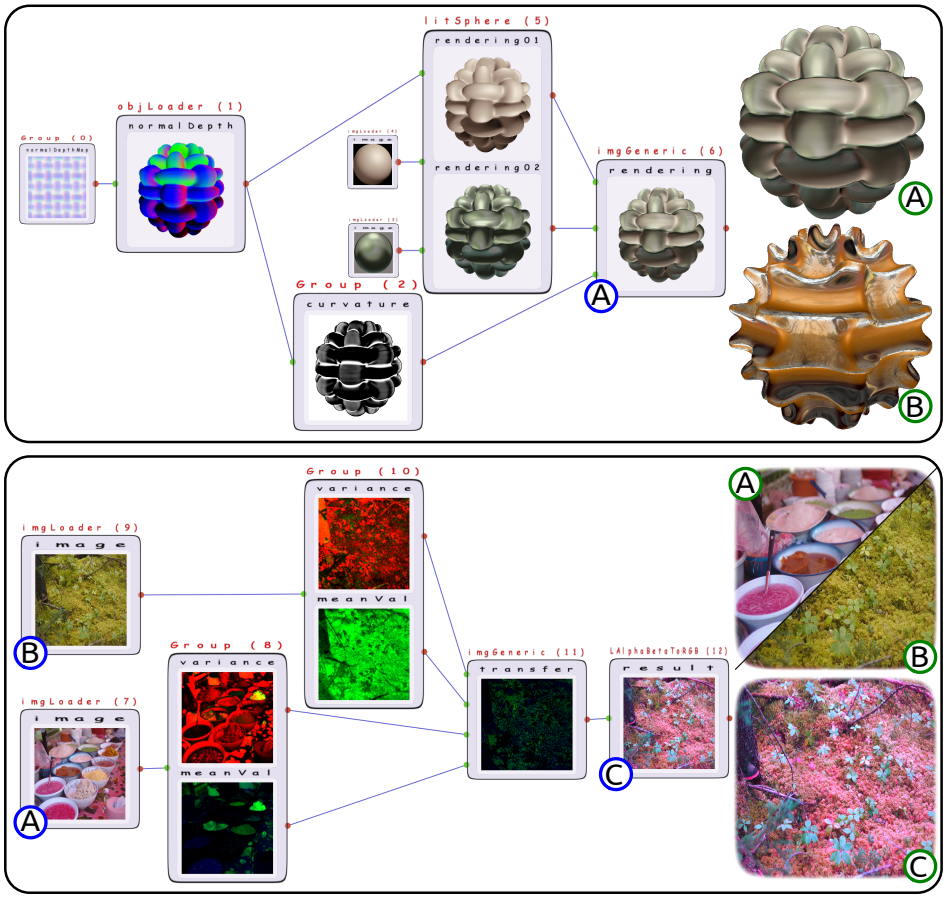


Figure 1. Our system is tailored to the fast prototyping of graphics pipelines in real time on GPUs. Top: a displacement mapping applied to an input sphere, resulting in a 3D object. The images in (A) and (B) are obtained by mixing two shading effects based on surface curvature, with different parameters. Bottom: a color-transfer pipeline that applies the color palette of a target image (A) to a source image (B), using global mean and variance properties. The image in (C) is obtained by directly writing a GLSL transfer function from inside the UI.

 ISSN 2331-7418

Abstract

Nodal architectures have received an ever-increasing endorsement in computer graphics in recent years. However, creating a node-based system specifically tailored to GPU-centered applications with real-time performance is not straightforward. In this paper, we discuss the design choices we took in the making of Gratin, our open-source node-based system. This information is useful to graphics experts interested in crafting their own node-based system working on the GPU, either starting from scratch or taking inspiration from our source code. We first detail the architecture of Gratin at the graph level, with data structures permitting real-time updates even for large pipelines. We then present the design choices we made at the node level, which provide for three levels of programmability and, hence, a gentle learning curve. Finally, we show the benefits of our approach by presenting use cases in research prototyping and teaching.

1. Introduction and Previous Work

Turning a graphics-related idea into working software is time-consuming and skill-demanding. It means developing a program architecture that works on the GPU, yielding pipelines that need to combine multiple passes. For non-experts, like artists or students, the steep learning curve might be discouraging, while for graphics experts, the process is tedious at best.

Such complex graphics pipelines have been handled with great success using node-based interfaces in the shading and animation workflows of 3D software such as Maya, 3DSMax, or Modo. Nodal solutions even form the core of highly-acclaimed programs such as Nuke for compositing and visual effects, Substance for texturing, or vvvv for interactive multimedia installations. Their broad endorsement is explained by their versatility, modularity, efficiency, intuitive visualization, and fast learning curve. Even though they incorporate GPU-accelerated nodes, these systems are not tied to specific graphics-card architectures.

Another way to learn or prototype graphics ideas is to use WebGL systems such as Shadertoy or GLSL Sandbox which allow users to edit a fragment shader and observe results with direct feedback in a separate viewer. More sophisticated applications also include programmable vertex shaders (Shdr) and provide multiple OpenGL setting choices, as in kickjs. These editors are limited to simple shaders though and do not allow for editing of other graphics stages, like tessellation or geometry shaders, or design complex, multi-pass pipelines. A notable exception is the ShaderTool software that allows users to edit shaders in a node-based interface similar to vvvv and to create multi-pass pipelines. This system mainly targets FX designers to easily prototype video-game shaders. It is not free nor open source, which does not make it easily adapted for teaching or research applications. Moreover, its does not handle complex animations and, to our knowledge, no API is provided to implement specific nodes such as multi-grid solvers or custom mipmaps.

Because of these limitations, we decided to develop our own node-based system tailored to the creation, manipulation, and animation of 2D/3D data in real time on GPUs. Our software is called Gratin, and it targets not only FX designers, but also computer graphics researchers and teachers: it offers three levels of programmability which make it suitable for the needs of experts and non-experts alike. Gratin is written in C++ and uses Qt for the interface, Eigen for linear algebra, OpenGL for renderings, and, optionally, OpenExr for loading and saving high dynamic range images. It is free and open source (licensed under MPL v2.0) and relies on OpenGL and GLSL to ensure wide OS and GPU compatibility. Source code and installation packages are available at `http://gratin.gforge.inria.fr/`.

The purpose of this paper is to describe the *design* decisions we took in the making of Gratin. These decisions should be useful to graphics experts interested in crafting their own node-based system working on the GPU, either starting from scratch or taking inspiration from our source code. We begin with a quick presentation of the user interface of Gratin in Section 2, which will help relate subsequent illustrations to the structure of the software. One key aspect of a node-based system is the graph architecture. In particular, it should provide fast updates, which is facilitated by the use of topological lists as explained in Section 3. Another important quality of such a system is ease of use. To this end, we propose three types of nodes (groups, generics, and plugins), as detailed in Section 4, which provide customization abilities for users with different skills. The practical usage of Gratin is demonstrated in Section 5, where we show a few examples in research prototyping and teaching that serve as proof of concept for our nodal design.

2. User Interface

As shown in Figure 2, the interface of our system is composed of five main panels.

- The pipeline panel (a) where one can interactively add, connect, disconnect, copy, or paste nodes. Node outputs are visualized in real time inside the interface, as in the (discontinued) nodal shading system Mental Mill.
- All available nodes are stored in user-defined directories and automatically loaded in the node tree (b) at initialization.
- The viewer (c) allows for the display of particular node outputs and for their manipulation via keyboard or mouse events.
- Each node has its own user interface that can be displayed and manipulated via a list of widgets (d).
- Any node parameter may be keyframed and interpolated via control curves in the animation panel (e).

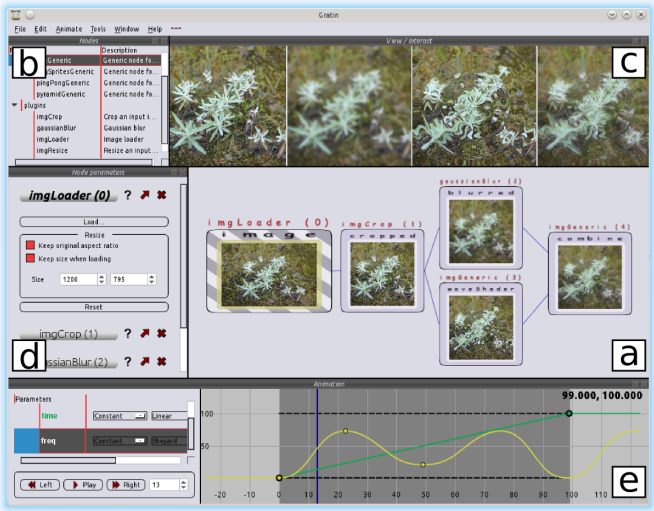


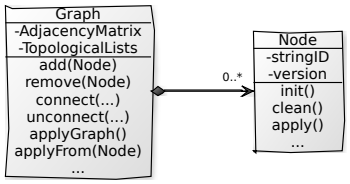
Figure 2. Gratin Interface. (a) Graph visualization; (b) list of available nodes; (c) node viewer; (d) node interfaces; (e) animation parameters and curves.

3. Architecture

Node types. When designing a node-based system, the first questions one may ask are to what each node of the graph corresponds and what type of data travels along edges of the graph. Some software such as vvvv allows for a huge range of node types and data that may flow along connections: even float parameters are represented as nodes. Other software such as Nuke more strongly restricts node types (e.g., image filters) and the data (e.g., images), while parameters are exposed through an additional panel in the interface. The main reason we opted for the latter design solution is that it better fits the architecture of modern graphics cards. Each node encapsulates one (or more) GLSL shader(s) that may have multiple inputs, outputs, and user-defined parameters. Graph edges hold 2D textures in OpenGL RGBA_32F format. This basic structure is directly adapted to multiple passes, which simply amount to connecting nodes one after the other. The choice of texture format permits the use of various types of data: from color images (including HDR) to height fields or normal maps. Data stored on edges could be extended to other types of GPU buffers. Nevertheless, we've found this design choice to be a good starting point as it allows users to work with 3D inputs through the use of g-buffers [Saito and Takahashi 1990] (textures holding geometric data), as demonstrated in the remainder of the paper.

Performance. An important practical aspect of a node-based system is performance; in particular, transfers between CPU and GPU should be minimized. In our case, this is ensured by mapping all nodes directly to GPU shaders and making sure that data held on edges *stay* on the graphics card. Moreover, the viewer and graph panels of

the interface are entirely written in OpenGL, with simple GLSL fragment shaders, to avoid unnecessary transfers from GPU to CPU. In practice, the output textures of each node are automatically attached to a dedicated framebuffer object during an initialization stage. This allows the use of custom shaders that directly render into the corresponding multiple output targets.



Graph structure. For the graph itself, we use a directed acyclic multigraph that is internally represented with an adjacency matrix [Cormen et al. 2009]. Its main advantage is to grant access to input and output nodes in constant time. However, this matrix alone is not adapted to ensure that nodes be processed in the right order when one needs to update and propagate information along the graph (topological ordering is known as a $\mathbf{NC}^2$-complex problem). Such updates may occur in two situations: (1) when making use of animation curves to control node parameters through time, which possibly impacts the entire graph; (2) when modifying the parameters or shader code of a single node, which will impact only neighbor nodes in the graph. In both cases, we make use of topological lists to ensure fast updates in the right order (see Figure 3). For animations, we maintain a topological list of the entire graph, computed using a depth-first search algorithm. This permits updates in linear time, with a node being refreshed only after all nodes able to reach it have been updated. Of course, only nodes for which parameters have changed (or those affected by them) are updated to avoid useless processing. For node editing, we maintain a topological list per node so that only reachable neighbor nodes are processed. As shown in the inset, all topological lists are held in the graph class and updates are all managed by that class. In particular, when adding or removing nodes and connections, the adjacency matrix and topological lists must be recomputed. We take advantage of this small lag to initiate GPU data in newly available nodes (memory allocation, shader initialization, framebuffer object creation, etc).

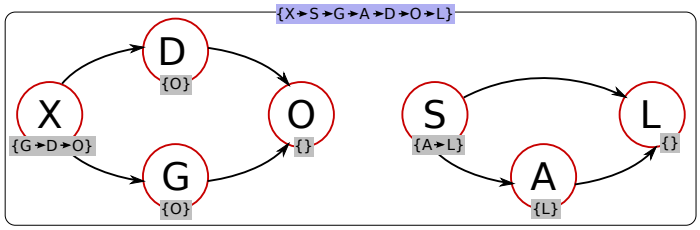


Figure 3. We maintain two types of topological lists: one at the graph level (in blue), used for animation, and one per node (in grey) used when its parameters or shader code is edited. Note that, in general, a topological list is not unique as different orderings may be valid.

Extensibility. Nodal architectures present a decisive advantage in the long term: new nodes may be added, bringing new features without having to upgrade the central architecture. This makes node-based systems easily extensible and, to some extent, customizable as different users may prefer to use different sets of nodes. In order for this to work in practice, it requires that new nodes do not break previously created pipelines; in other words, it must be backward compatible. To this end, we store pipelines as XML files, where each node is identified by a string ID and a versioning number to prevent input/output issues. With this approach new plugins and user-defined custom nodes can easily be added to the system and stored in different directories as soon as they are specified to the system. If a node is upgraded in a way that no backward compatibility can be ensured, its old version can be stored somewhere else in the list of nodes so that previous pipelines will still be able to load without any errors. The range of GPUs supported by a given node is implicitly restricted by the GLSL version with which it is compatible. When running Gratin on a computer configuration that does not support the correct version, we leave the possibility of modifying this version by hand in the settings window. This solution will of course only work in cases where the node is actually compatible with the modified GLSL version.

4. Node Design

One of the most valued properties of nodal systems is their modularity: one may combine nodes, customize their behaviors, or write entirely new nodes from scratch. Each of these node-design strategies bears different names in existing nodal systems; in Gratin, we call them group, generic, and plugin nodes, respectively. They represent three levels of programmability essential to providing a gentle customization abilities for all users.

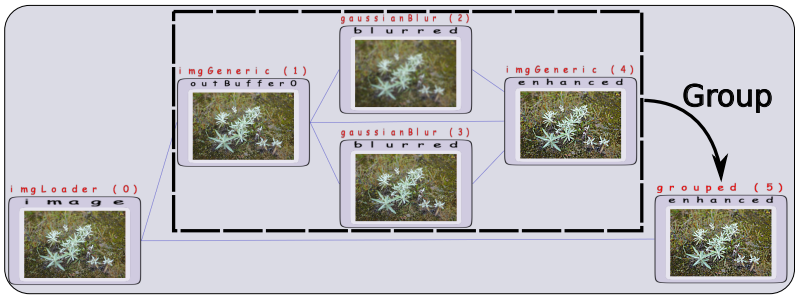
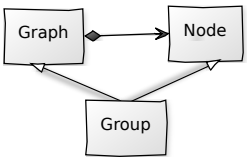


Figure 4. Grouping. Selected nodes represented with the dashed rectangle are bundled into a single group node (bottom right) that produces exactly the same results.

4.1. Group Nodes

Groups simply combine multiple nodes into a single one (see Figure 4), hence providing users with a means to design new functionalities without requiring any knowledge of programming. A group node has its own interface that contains widgets of all internal nodes so that users may still modify their parameters. It not only improves the legibility of a graph (especially when copy-pasting different instances of a group), but also simplifies the exchange of custom features among users.

In practice, a group is an object that inherits the graph and the node classes as shown in the inset image. When a group is created, selected nodes (the subgraph) are automatically inserted into a new directed acyclic multigraph that can itself be interpreted as a single node to be processed. This way, a group can be displayed and manipulated like every other node, while implicitly updating its whole graph (using the internal topological list) when a refresh is needed. Input and output entries are extracted and stored in wrappers to ensure fast connections. An advantage of having group nodes inheriting both from Node and Graph classes is that they may be recursively created, meaning that groups may contain other groups and so on. When saved, group nodes are exported as XML files. They may then trivially be included in the list of nodes for instant reuse.



4.2. Generic Nodes

Generic nodes provide users with the ability to precisely customize processes to their needs by writing GLSL shaders directly from inside the user interface. The creation of a generic node begins with the filling of a dialog box where users specify important properties of a GLSL shader (see Figure 5). These include OpenGL specifications (e.g., depth test, blending, texture filtering, and wrapping), shader choices (vertex, tessellation, geometry and/or fragment shaders, GLSL version), number and

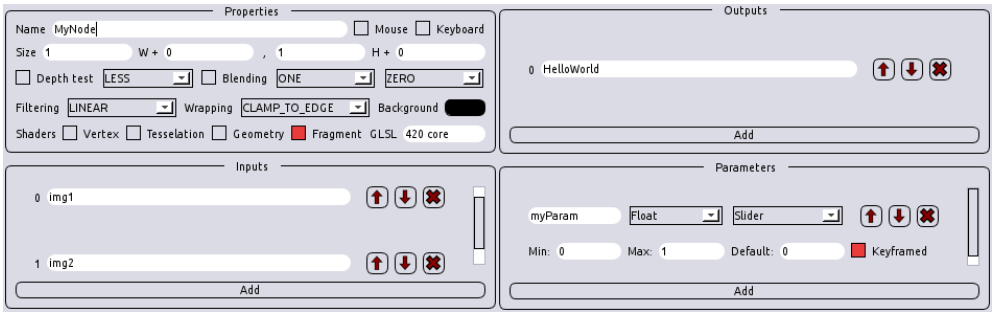
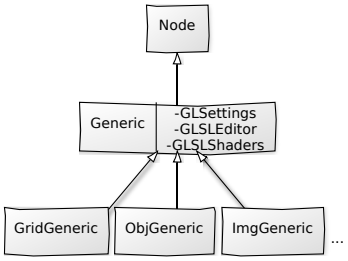


Figure 5. Generic settings. Multiple parameters, inputs and outputs might be chosen to initialize OpenGL settings and shader headers.

name of input and output textures, and an arbitrary number of controllable parameters (uniform variables) sent to the GPU program. Once the dialog is validated by the user, each shader parameter is assigned a widget depending on its type, and an empty shader template is generated with the *proper header*. Users may then customize the node behavior by writing GLSL code and observing results in real time.

Even though we provide a family of generic nodes that have their own specificities, they are all based on the same design principle. As shown in the inset, they inherit a parent Generic class that holds settings users have made through the dialog box. This parent class also handles the GLSL editor as well as the initialization of GLSL shaders, which simplifies the design of new types of generic nodes. As with group nodes, customized generic nodes may be exported, exchanged and inserted in the list of nodes to be reused in other pipelines.



In the following, we present the different types of generic nodes currently available in Gratin, in each case explaining how shader headers are generated. As will soon be apparent, each type of generic node offers a process commonly used in the graphics community, hence encapsulating technical know-how of GPU programming.

The generic image node permits the user to analyze, manipulate, and visualize input textures (e.g., colored images, g-buffers). As is commonly done for image processing in OpenGL, a simple quad is drawn in the viewport so that input textures can be easily accessed and mapped to create outputs containing specific effects. This node may be used to create one-pass custom complex shaders (as in Shadertoy for instance), to mix input images, to create various patterns, etc. Figure 6 provides an example where we select the maximum color between two textures. Implementation-wise, user-specified GLSL version, input and output texture names are automatically used to generate the header of the shader. By default, the fragment shader simply makes a copy of the first

```
// part automatically generated with user-defined settings
#version 420 core
layout(location = 0) out vec4 maxColor;
uniform sampler2D img1;
uniform sampler2D img2;

// part written / modified by the user
in vec2 texcoord;
void main() {
  vec4 c1 = texture(img1,texcoord);
  vec4 c2 = texture(img2,texcoord);
  maxColor = max(c1,c2);
}
```

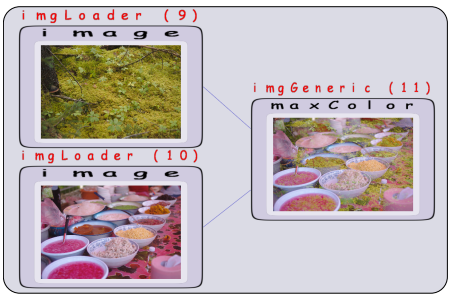


Figure 6. Left: The header of the fragment shader is automatically generated depending on user settings (inputs, outputs, parameters, etc). The main function might be directly modified to obtain the desired effect. The maximum color is displayed here, as shown on the right.

input texture. The example in Figure 6 thus simply consists in modifying a couple of lines of code.

Such an approach is reminiscent of Expression nodes in Nuke, which allow the application of complex formulae to colors of input images. The writing of GLSL code is more general though, as it builds from an extensive list of operators and functions. In our experience, we also use it to implement basic image-blending operators such as additions or multiplication, instead of providing specific built-in nodes.

The generic object and grid nodes let users apply any effect to 3D meshes by customizing vertex, tessellation, geometry, and fragment shaders. The main difference between object- and grid-node types is that the former loads a mesh in OBJ format, while the latter creates a planar grid. In the case of the grid node, tessellation is chosen by the user in the dedicated interface; vertices are then typically displaced according to input GLSL code. As with other generic nodes, any number of textures might be provided as input (such as color or normal maps for instance). Outputs are typically in the form of g-buffers or renderings for further 3D or 2D processing, respectively. A trackball camera is associated to this node so that users may manipulate their object in the viewer panel.

```
// part automatically generated with user-defined settings
#version 420 core
layout(location = 0) in vec2 inVertex;
layout(location = 1) in vec2 inTexcoord;
uniform mat4 model;
uniform mat4 view;
uniform mat4 proj;
uniform float a; // user-defined parameter
uniform float b; // user-defined parameter

// part written / modified by the user
out vec3  normalV;

vec4 evalFunc(in float x,in float y) {
  // return depth+normal
  float f = a*x*x+b*y*y;
  float fx = 2.*a*x;
  float fy = 2.*b*y;
  return vec4(f,normalize(vec3(-fx,-fy,1.)));
}

void main() {
  mat4 mdv = view*model;
  mat4 mvp = proj*mdv;
  vec4 f   = evalFunc(inVertex.x,inVertex.y);
  vec3 v   = vec3(inVertex.xy,f.x);
  normalV  = (mdv*vec4(f.yzw,0)).xyz;
  gl_Position = mvp*vec4(v,1);
}
```

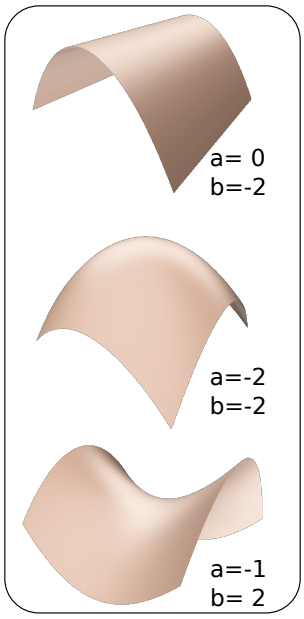


Figure 7. A vertex shader designed to visualize a quadratic function (left). As seen in the code, default attributes (vertices, texture coordinates) as well as camera matrices are automatically sent to the GPU. Two variables (a and b) were also defined by the user and controllable in real-time using integrated sliders inside the interface of the node. The renderings (right) are obtained via a simple fragment shader. Values for a and b are shown for each of them.

An example of a generic grid node is given in Figure 7. It visualizes a quadratic function; the grid node is, of course, also useful to create terrains or other types of heightfields. In practice, the mesh (either loaded OBJ or grid) is sent to the GPU using vertex array objects. In both cases, mesh positions, normals, tangents, and texture coordinates are directly sent as vertex attributes. Consequently, the header of the vertex shader is adapted to grant access to these attributes, as seen in the left part of the figure. We also show an example of a generic object node as a proof of concept of our system in Figure 12.

The generic splat node permits the manipulation of point sprites. We found this node useful to control particles, visualizations, or even image warpings. Indeed, the particularity here is to be able to modify splat sizes and locations (possibly using overlapping and blending) to obtain specific effects. The interface allows the control of the number of rendered sprites, and their behavior is controlled through GLSL code. In practice, it works by sending a set of point sprites to the GPU, with one splat-per-pixel by default. Figure 8 shows an example where two input renderings are compared using joint histograms. It involves a geometry shader that is invoked three times (which triples the number of splats) and compares each input color channel to reposition displayed points. Shader headers for the generic splat node contain the position of the splat (as an attribute in the vertex shader), as well as uniform variables. The remainder of the shader can be freely modified by users.

```
// part automatically generated with user-defined settings
#version 420 core
uniform sampler2D img1;
uniform sampler2D img2;

// part written / modified by the user
layout(points, invocations = 3) in;
layout(points, max_vertices = 3) out;

out vec4 color;

void main() {
  vec2 coord  = gl_in[0].gl_Position.xy*.5+.5;
  vec3 color1 = texture(img1,coord).xyz*2.-1.;
  vec3 color2 = texture(img2,coord).xyz*2.-1.;
  int id      = gl_InvocationID;
  color       = vec4(vec3(0.),.2);
  color[id]   = 1.;
  gl_Position = vec4(color1[id],color2[id],0.,1.);
  EmitVertex();
  EndPrimitive();
}
```

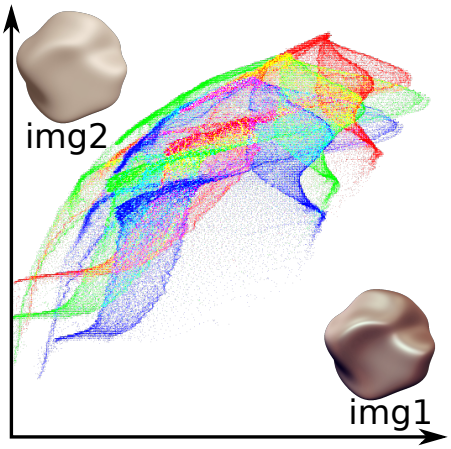


Figure 8. The geometry shader (left) generates splats to compare two renderings with joint histograms (right). Splats are repositioned so that their x- and y-coordinates correspond, respectively, to pixel intensities of the first and second input images. The fragment shader simply displays the output color in this example.

```
// part automatically generated with user-defined settings
#version 420 core
layout(location = 0) out vec4 minColor;
layout(location = 1) out vec4 maxColor;
uniform sampler2D img;          // input image
uniform sampler2D prevLevel0; // previous (minColor) level
uniform sampler2D prevLevel1; // previous (maxColor) level
uniform int nbLevels;
uniform int numLevel;
uniform bool isFinest;
uniform bool isCoarsest;

// part written / modified by the user
in vec2 texcoord;

void loadVals(in sampler2D tex,out vec4 c[4]) {
  vec2 ps = .5*(1./vec2(textureSize(tex,0)));
  c[0] = texture(prevLevel0,texcoord-ps);
  c[1] = texture(prevLevel0,texcoord+ps);
  c[2] = texture(prevLevel0,texcoord+vec2(ps.x,-ps.y));
  c[3] = texture(prevLevel0,texcoord+vec2(-ps.x,ps.y));
}

void main() {
  if(isFinest) {
    minColor = maxColor = texture(img,texcoord);
  } else {
    vec4 a[4];
    vec4 b[4];
    loadVals(prevLevel0,a);
    loadVals(prevLevel1,b);
    minColor = min(min(min(a[0],a[1]),a[2]),a[3]);
    maxColor = max(max(max(b[0],b[1]),b[2]),b[3]);
  }
}
```

Figure 9. Fragment shader of a generic pyramid node (left) used to compute two output pyramids containing the min and max colors of an input image. A set of predefined uniform variables are automatically sent to the shader to provide information about the pyramid (the total number of levels, the current level, etc). *prevLevel0* and *prevLevel1* represent previously computed levels for the min and max output pyramids. The visualization (right) was done in a simple generic image node by accessing the different levels of the two pyramids.

The generic pyramid node creates one or more mipmaped textures where one can control how each level is computed. It might be used for the creation of usual mipmaps, but also for multiscale analysis (Gaussian or Laplacian pyramids) or to compute global information such as the mean and variance of an input image (see Figure 9).

On the implementation side, we automatically generate a header that contains variables, such as the number of levels and flags,in order to know whether the top or bottom of the pyramid has been reached. In addition, the previously computed level of each input texture is automatically added to the GPU program as a uniform sampler. Users, thus, only have to describe per-level operations in a specific order (top-down or bottom-up) in the GLSL code. The resulting pyramids are stored as mipmaps, which allows us to leverage existing texture-access operators from OpenGL. Further connected nodes may thus easily access any texture level via GLSL built-in functions, such as *textureLod()*.

```
// part automatically generated with user-defined settings
#version 420 core
layout(location = 0) out vec4 filtered;
uniform sampler2D img;         // input image
uniform sampler2D pingpong0;   // previous filtered image
uniform int numpass;           // current pass

// part written / modified by the user
in vec2 texcoord;

void loadVals(in sampler2D tex,out vec3 vals[5]) {
  // neighborhood
  vec2 ps = 1./vec2(textureSize(tex,0));
  vals[0] = texture(tex,texcoord).xyz;
  vals[1] = texture(tex,texcoord+vec2(-ps.x,0.)).xyz;
  vals[2] = texture(tex,texcoord+vec2(ps.x,0.)).xyz;
  vals[3] = texture(tex,texcoord+vec2(0.,-ps.y)).xyz;
  vals[4] = texture(tex,texcoord+vec2(0.,ps.y)).xyz;
}

float dist(in vec3 c1,in vec3 c2) {
  // distance control
  float r = 0.01;
  float o = 0.02;
  float d1 = (c1.x+c1.y+c1.z)/3.;
  float d2 = (c2.x+c2.y+c2.z)/3.;
  return 1.-smoothstep(o-r/2.,o+r/2.,abs(d1-d2));
}

vec4 applyFiltering(in sampler2D tex) {
  // anisotropic filtering
  vec3 Vf[5];
  vec3 Vd[5];
  loadVals(tex,Vf);
  loadVals(img,Vd);

    vec3 R = Vf[0] +
      .1*(dist(Vd[1],Vd[0])*(Vf[1]-Vf[0]) +
        dist(Vd[2],Vd[0])*(Vf[2]-Vf[0]) +
        dist(Vd[3],Vd[0])*(Vf[3]-Vf[0]) +
        dist(Vd[4],Vd[0])*(Vf[4]-Vf[0]));

  return vec4(R,1.);
}

void main() {
  if(numpass==0) { // first pass: copy input
    filtered = texture(img,texcoord);
  } else { // otherwise: successively apply filtering
    filtered = applyFiltering(pingpong0);
  }
}
```

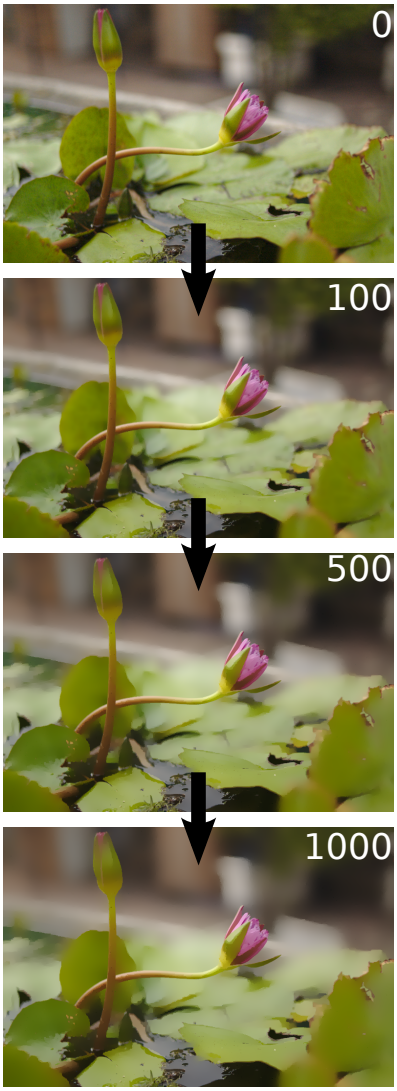


Figure 10. This fragment shader is iteratively applied to a texture to perform an anisotropic filtering as shown on the right. The previous texture, filtered with this same shader, is available as a uniform sampler (*pingpong0*) as well as the number of the pass. In this example, the first pass simply copies the input image, which is then filtered by a user-defined number of passes.

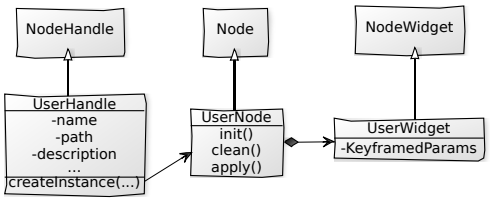
The generic ping-pong node provides another useful feature commonly used in GPU programming to implement iterative processes. The same process is applied at each internal pass and repeated the number of times specified by the user. Such a type of node may be used to iteratively accumulate or propagate some information in the output textures. An example is shown in Figure 10 where multiple passes are applied in a ping-pong node to obtain a controllable anisotropic filter.

In practice, it uses a pair of textures for internal multiple passes, with one texture being read and the other written on even passes, and the opposite on odd passes. However, this is not apparent to the user: we automatically generate a header that gives access to the resulting texture of the previous internal pass, as well as the current pass number. Note that such a ping-pong architecture could not be created manually by connecting simpler nodes, since our graph is acyclic.

4.3. Plugin Nodes

Plugins represent the most advanced way to design new nodes. They must be compiled beforehand and cannot be modified in real time as with generic nodes. They are thus more adapted to the implementation of features that demand specific internal node architectures. For instance, Poisson diffusion requires a multigrid architecture, and fast Fourier transforms require multiple iterations and specific butterfly tables that cannot be easily obtained through the use of generic nodes.

To ease the development of plugin nodes, we aimed for a light-weight C++ API augmented with helper classes and functions (see inset). A plugin consist of at least a pair of classes, inheriting of NodeHandle and Node classes. An additional class inheriting from NodeWidget may be defined to deal with parameters.



The role of the node handle is to make the association between the user interface and the node (Figure 11(left)). It contains basic information such as the node name, description, help, and input/output names. The node Id and version are also specified in this class in order to ensure that the node be unique and reusable in any pipeline. Most importantly, this class is responsible for creating instances of the node itself. Indeed, the main program contains the list of all available handles (exposed to users in the node tree): each handle creates the instance of the corresponding node and adds it to the current pipeline when requested by the user.

The role of the node is to actually process the data (Figure 11(right)). It contains operators such as *init()*, *clean()*, or *apply()* that can be overloaded to process shaders using input data and parameters. We also provide helper classes to simplify GPU initialization and transfers inside the node. Framebuffer objects can automatically be created and associated with the output textures when the node is ready to be applied. Tools are made available to load, compile, and link shaders from a set of files, and built-in functions might be used to create buffers and draw any mesh given a shader program. Predefined parameters are provided to quickly add and modify parameters to the optional node widget panel. Taken together, this set of classes, functions, and

```
class MyHandle : ..., public NodeHandle {
 // ... QT plugin information ...
 public:
  // node version and ID
  unsigned int version  () const
    {return 1;}
  const QString stringId() const
    {return "myNodeID";}

  // basic node information
  const QString name    () const
    {return "myNode";}
  const QString path    () const
    {return "plugins/";}
  const QString desc    () const
    {return "my node description";}

  // inputs and outputs (1 input, 2 outputs)
  const QStringList inputNames () const {
    return QStringList() << "inTex"; }
  const QStringList outputNames() const {
    return QStringList() << "outTex1"
                         << "outTex2";}

  // create an instance of MyNode
  Node *createInstance(Graph *g) {
    return new MyNode(g,new NodeHandle(this));
  }
  // ...
};
```

```
class MyNode : public Node {
 public:
  MyNode(Graph *g,NodeHandle *h) : Node(g,h),
    _p("shader/files") {
    _p.addUniform("myTex");
  }

  void init () {/* called when connected    */}
  void clean() {/* called when disconnected */}

  void apply() { /* runtime */
    // set viewport to texture size
    setViewport(outputTex(0)->w(),
                outputTex(0)->h());

    // enable GPU settings
    _p.enable();
    _p.setUniformTexture("myTex",inputTex());

    // multiple render targets
    drawOutputs(buffersOfOutputTex(0),
                nbOutputs());

    // disbale GPU settings
    _p.disable();
  }

 private:
  GPUProgram _p;
};
```

Figure 11. Left: The node handle contains node information such as ids, name, description, input and output names, etc. The *createInstance* function will be called whenever users add this node to the graph. Right: The node itself overloads basics functions such as *init*, *clean*, or *apply*. In this example, the node sends the input texture to the shader program and automatically applies a multiple render target to draw into the two output textures. Note that for better clarity, some function and class names have been slightly modified here.

tools make it possible to write a simple plugin using only a few lines of code, as demonstrated in Figure 11.

5. Discussion and Future Work

The design of an efficient node-based system working on the GPU requires a careful architecture organization. The solution chosen for our system ensures fast pipeline updates through the use of topological lists. However, the total performance of a pipeline depends on the complexity of each node. While most nodes work at real-time frame-rates, some computations are more time demanding, as is typically the case with the ping-pong node for instance. In such situations, our strategy has been to include an "apply" button, so that users may control when to update the node output.

The learning curve is another important concern when designing a node-based system. Our approach has been to provide three levels of programmability through three types of nodes (groups, generics, and plugins). We believe that these design choices will allow users from different backgrounds to quickly create new graphics

applications. Of course, each level of programmability provides the ability to create increasingly complex effects: what can be achieved with a plugin is hardly done with a group.

In the following, we provide example uses of Gratin for research prototyping and teaching in graphics. These are proofs that a GPU-tailored node-based system such as our own can accommodate a diversity of applications.

Prototyping. One of the direct benefits of our generic node design is to quickly provide a way to test and combine existing GPU code with simple copy-paste. This is illustrated in Figure 12: the top row shows the Phong tessellation technique [Boubekeur and Alexa 2008] applied to an input icosphere and rendered in wireframe using Baerentzen et al.'s technique [Bærentzen et al. 2006]. The bottom row shows the flame shader of Shadertoy, integrated in a generic image node, where the time was included as a keyframed animated parameter. Both examples were created in a few minutes, the time to set up the proper widgets for the copy-paste code.

A nodal system also permits to quickly test promising graphics ideas and experiment with various alternatives that mix 2D and 3D data. For instance, an early version of Gratin has been used for the implementation of the surface flows technique [Vergne et al. 2012] [1], making use of dedicated plugin nodes. This is not limited to computer graphics research, as Gratin has been employed to create stimuli for perceptual experiments [Fleming et al. 2013; Dovencioglu et al. 2015]. We have also been contacted by researchers in graphics-demanding fields such as cartography, molecular biology, and archeology who are interested in controlling the image-creation process and comparing different results.

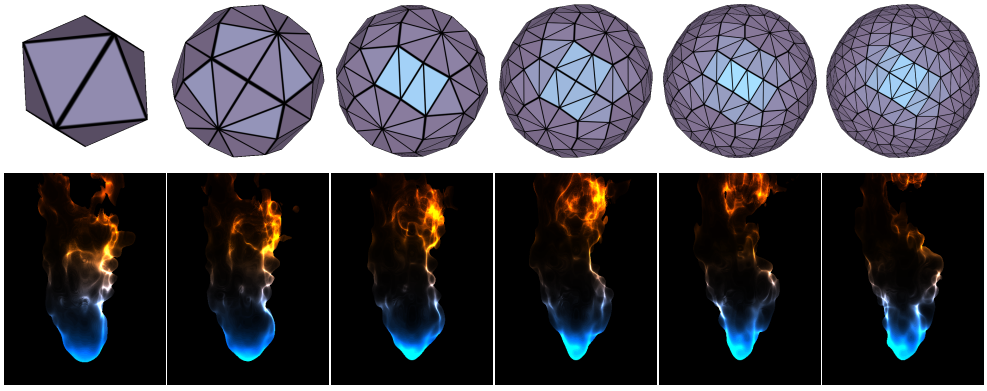


Figure 12. Top: Phong tessellation implemented via evaluation and control shaders. The tessellation level is controlled in real ime with a user-defined slider. The rendering is designed in geometry and fragment shaders. These images result from a single generic object node. Bottom: Flame shader copied from Shadertoy and pasted in a generic image node.

[1] See http://vimeo.com/44439181.

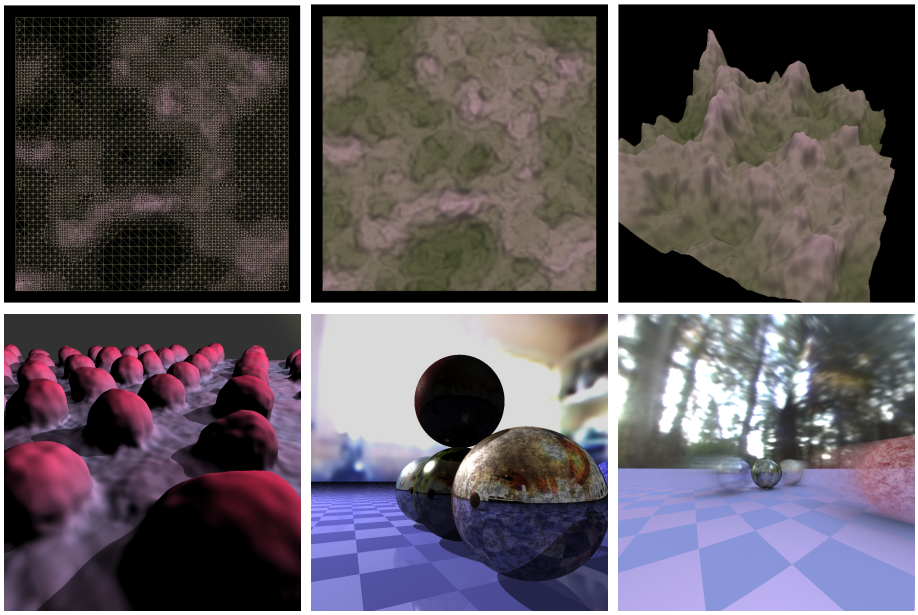


Figure 13. Top: tessellation done in a generic grid node for rendering a terrain with an increasing number of triangles on curved regions. Bottom - from left to right: ray-casting (generic image node), ray-tracing (generic image node) and path-tracing (ping-pong node).

Teaching. We believe our design choices are well adapted to education purposes, since Gratin allows students to learn GPU programming by focusing on GLSL without having to worry about the OpenGL architecture. In our experience, students typically start by grouping existing nodes, then customizing generic nodes, and eventually writing their own plugins if necessary. Figure 13 illustrates examples of assignments given to students in an advanced image synthesis course. In the top row, students were asked to displace and adaptively tessellate a grid depending on given input depth, normal and curvature maps. The bottom row shows examples where students had to implement ray-casting on implicit surfaces using ray-marching (left) and ray-tracing with fractal Perlin noises (center) using generic image nodes. The last example (right) is a path-tracer entirely written in the fragment shader of a single ping-pong node. Last but not least, the nodal structure itself with its real-time visualization and interactive controls is a great support for teaching as well as dissemination of research.

5.1. Future Work

Although the conception of Gratin makes it adapted to various application contexts, there are still limitations on which we plan to work in the future. Most importantly, current pipelines exclusively exchange 2D textures. Even if other types of data can be manipulated internally in specialized nodes (3D meshes, 3D textures, etc.), it might be

useful to transfer them between nodes in order to design, chain, and combine shaders working on different kinds of GPU buffers. Another limitation concerns parameters that are dedicated to each shader and cannot be currently linked between nodes. For instance, the parameters of a camera cannot be currently shared to simultaneously control two scenes. Our viewer is also quite basic in the current version, and we would like to extend it to ease prototyping. To this end, we could add a tone mapper and channel selector, pixel information when hovering the mouse on a node output, as well as frame-rates for both the entire pipeline and viewed node outputs. Finally, we plan to provide new generic node designs, such as one based on depth peeling, which could be useful for transparency effects. Other techniques requiring novel generic node designs include shadow mapping or motion/lens blur, for instance. Implementing these additional features will open our nodal-based system to even more diverse applications.

Acknowledgements

We gratefully acknowledge the anonymous reviewers for their helpful and positive comments. We thank Paul Debevec for the light probes used to generate the two bottom-right images in Figure 13. We also thank Thomas Hurtut for the input photos used in Figures 1, 4, 6, 9, and 10.

References

BÆRENTZEN, A., NIELSEN, S. L., GJØL, M., LARSEN, B. D., AND CHRISTENSEN, N. J. 2006. Single-pass wireframe rendering. In *ACM SIGGRAPH 2006 Sketches*, ACM, New York, NY, SIGGRAPH '06. URL: http://doi.acm.org/10.1145/1179849.1180035, doi:10.1145/1179849.1180035. 68

BOUBEKEUR, T., AND ALEXA, M. 2008. Phong tessellation. *ACM Trans. Graph. 27*, 5 (Dec.), 141:1–141:5. URL: http://doi.acm.org/10.1145/1409060.1409094, doi:10.1145/1409060.1409094. 68

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2009. *Introduction to Algorithms*, 3rd ed. The MIT Press, Cambridge, MA. 58

DOVENCIOGLU, D. N., WIJNTJES, M. W., BEN-SHAHAR, O., AND DOERSCHNER, K. 2015. Effects of surface reflectance on local second order shape estimation in dynamic scenes. *Vision Research*. 68

FLEMING, R., VERGNE, R., AND ZUCKER, S. 2013. Predicting the effects of illumination in shape from shading. *Journal of Vision 13*, 9, 611. URL: http://www.journalofvision.org/content/13/9/611.abstract, doi:10.1167/13.9.611. 68

SAITO, T., AND TAKAHASHI, T. 1990. Comprehensible rendering of 3-D shapes. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, SIGGRAPH '90, 197–206. URL: http://doi.acm.org/10.1145/97879.97901, doi:10.1145/97879.97901. 57

VERGNE, R., BARLA, P., FLEMING, R., AND GRANIER, X. 2012. Surface Flows for Image-based Shading Design. *ACM Transactions on Graphics 31*, 3 (Aug.), 94:1–94:9. URL: http://hal.inria.fr/hal-00702280, doi:10.1145/2185520.2185590. 68

Author Contact Information

Romain Vergne
INRIA Rhône-Alpes
655 avenue de l'Europe
38334, Saint Ismier Cedex FRANCE
romain.vergne@inria.fr
http://maverick.inria.fr/~Romain.Vergne

Pascal Barla
INRIA Bordeaux Sud-Ouest
200, av. de la Vieille Tour
33405 Talence, France
pascal.barla@inria.fr
http://www.labri.fr/perso/barla

R. Vergne and P. Barla, Designing Gratin, A GPU-Tailored Node-Based System, *Journal of Computer Graphics Techniques (JCGT)*, vol. 4, no. 4, 54–71, 2015
http://jcgt.org/published/0004/04/04/

Received:	2015-02-26		
Recommended:	2015-09-02	Corresponding Editor:	Marc Olano
Published:	2015-11-19	Editor-in-Chief:	Marc Olano