STAG: Smart Tools and Applications in Graphics (2019)
M. Agus, M. Corsini and R. Pintus (Editors)

Yocto/GL: A Data-Oriented Library For Physically-Based Graphics

F. Pellacini G. Nazzaro E. Carra

Sapienza University of Rome

Figure 1: *Production scene from Disney's feature film Moana, rendered with Yocto/GL on a laptop. Our library focuses on code simplicity, feature composability and scalability to large environments, demonstrated in this rendering. Model courtesy of Disney Animation Studios [Stu18].*

Abstract

In this paper we present Yocto/GL, a software library for computer graphics research and education. The library is written in C++ and targets execution on the CPU, with support for basic math, geometry and imaging utilities, path tracing and file IO. What distinguishes Yocto/GL from other similar projects is its minimalistic design and data-oriented programming style, which makes the library readable, extendible, and efficient. We developed Yocto/GL to meet our need, as a research group, of a simple and reliable codebase that lets us experiment with ease on research projects of various kind. After many iterations carried out over a few years, we settled on a design that we find effective for our purposes. In the hope of making our efforts valuable for the community, we share our experience in the development and make the library publicly available.

CCS Concepts

• ***Computing methodologies*** → ***Ray tracing;*** *Mesh geometry models; Physical simulation; Graphics file formats;*

1. Introduction

Most research and education efforts in Computer Graphics use ad-hoc software, often developed by the same researchers that are investigating new topics. In this, our research field differs from the others, that have standardized and often industry supported software packages. For example, in Computer Vision it is common to use Matlab in conjunction with neural network software such as PyTorch. The reasons why graphics mostly uses hand-grown software are many, including historical and economical motivations, but also the sheer variety of research topics that we consider graphics.

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

DOI: 10.2312/stag.20191373

This paper describes Yocto/GL, a library for research and education in physically-based graphics. Briefly, Yocto/GL contains a low-level math library, utilities for computational geometry, utilities for tonal adjustments, a path tracer, and fast model IO. A previous version of the library also included an impulse-based rigid body solver and a PBD-style particle simulator, which are being rewritten. In terms of graphics features, each of these aspects is just a subset of known libraries, such as PBRT [PJH16], CGAL [The19] and Libigl [JP17], although Yocto/GL scales considerably better with model size. Figure 1 shows a movie-like scene rendered in Yocto/GL on a laptop.

We believe that what distinguishes Yocto/GL is the coding style and the overall software design we use in developing the library. This paper focuses on these two aspects, that we believe are applicable to other libraries and made a significant impact on our everyday use. Rather than adhering to abstract design decisions, we arrived at this design in a pragmatic manner, by re-implementing the same features multiple times using different programming styles until we reached good tradeoffs when using the library in our research and teaching efforts. In this paper, we cover the current design and motivate it with our experience, in the hope that this can be helpful to others invested in similar efforts.

We developed Yocto/GL since current libraries, including our previous efforts, did not support well our research. We focus on problems in content creation, where recent projects use features from computational geometry, physically-based rendering and animation. In general, most computer graphics libraries are complex, in terms of data structures and coding style. This complexity comes at the price of reduced flexibility since it is hard to compose features from different libraries without duplicating data and adopting different, often conflicting, programming styles. Also, we are interested in working with large data sets and found that many research libraries do not scale to scene size typical of real-world applications, both in terms of algorithms but also in how they are implemented. For practical reasons, we also wanted to use Yocto/GL when teaching graphics courses, to show students example implementations of algorithms and as base for homeworks and thesis. In summary, we needed a library that was *simple* to read and teach, *composable* to be adapted to different projects, and *efficient* to support low-latency applications and large environments.

Yocto/GL is written in C++ for efficiency and since most other libraries adopt that language. To make the library composable, we adopt a data-oriented, procedural, programming style rather than an object-oriented design. Data in Yocto/GL is explicit and directly accessible. Most APIs are just free functions that take explicit data as input. In fact, we use only a small handful of basic, generic, types that can be passed to many functions, rather than modeling problems by introducing many types and methods. To combat C++ complexity, we only use a subset of the language and do not use pointers, neither raw nor smart, relying instead on value semantics. All resources are stored in data-owning arrays and references are modeled as indices which refer to positions in the proper resource array. To our surprise, the resulting code is shorter, easier to read, scales very well with scene size, is more efficient to compute with, and has no memory corruption bugs while developing. Some of this design is vaguely similar to Libigl [JP17], from which we differ since we model whole scenes, heterogenous computations and use no template metaprogramming.

The main limitations of Yocto/GL come from our insistence on simplicity. We focus only on CPU computation and do not support GPU computation or even complex GPU viewers. GPU programming feels too cumbersome in modern APIs, like Vulkan, DX12 or Metal, and, more importantly, in our recent works we found that a progressive CPU raytracer scales better than a GPU rasterizer. We also insist on including external libraries rarely, and only if their APIs are simple, data-oriented in style, use little template metaprogramming, and they have no negative impact on build time or code readability. External libraries are always included with wrappers to ensure interoperability. These choices reduce the features we expose, but make the library much easier to use.

To this day, Yocto/GL has been quite successful in helping us reach our goals. We use the library in all recent research projects where it gave us a significant boost in productivity, especially in helping us tackle projects that are quite different in nature, but still rely on the same low-level functionality. We have also used Yocto/GL in teaching an undergraduate and a graduate course in graphics as well as several theses. Students informally reported that the difficulty of homeworks and theses were the graphics aspects, rather than the programming ones, which is surprising since our degree does not offer any course in C++ or high-efficiency programming.

2. Related Work

Reviewing all popular libraries for computer graphics research is out of the scope of this work. Instead, we recap here some such libraries mostly as comparison in terms of software design. We stress here that library design varies widely, and that all designs are valid because different tradeoffs come into play.

Physically-Based Rendering Pbrt [PJH16] and Mitsuba [Jak10] are the most used physically-based rendering frameworks in research. The aim of Pbrt is to give a reference implementation of a physically-based renderer from the ground up, comparing different materials models, geometry representations and various state-of-the-art rendering algorithms. The related Pbrt book provides a comprehensive discussion of modern rendering. Mistsuba was inspired by Pbrt, putting a stronger emphasis on experimental development of rendering techniques. Both libraries are written in an object-oriented style, thus reusing their code consists in the extension of base classes and the implementation of methods and interfaces. As discussed before, we drastically differ in programming style adopting a data-oriented design and favoring a fixed small set of shapes, materials and rendering methods.

Computational Geometry Libigl [JP17], CGAL [The19] and MeshLab [CCC*08] are the most used computational geometry libraries. Libigl focuses on usability with a programming style similar to Matlab and targets processing of triangle meshes. Rather than providing a full framework, Libigl defines many functions that act as building blocks for mesh processing. CGAL on the other hand, is a comprehensive collection of libraries for 2D and 3D mesh processing, with a focus on formal correctness. CGAL is also written as a toolset for writing computational geometry applications, but its

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

programming style makes it more suitable for final implementation than quick experimentation and teaching. Cinolib [Liv19] was developed with a similar purpose, but is specifically designed to support a wide variety of surface representations. A different approach is taken by MeshLab [CCC*08] which focuses on building a full-stack application with a usable interface, with a particular focus on support for 3D scanning workflows. Compared to these libraries, we support very few computational geometry operations which are mostly helpful in procedural modeling. Beside features, what we mostly differ on is that we adopt a significantly simpler programming style, avoiding almost entirely template metaprogramming and object orientation, both of which are the cornerstones of these libraries.

Physically-Based Animation Bullet [Cou10], Vega [SSB13], PositionBasedDynamics [Ben18a] and SPlisHSPlasH [Ben18b] are well-known animation libraries that differ in the type of simulation they support. Bullet is mostly an impulse-based rigid body solver used for special effects and simulation. Vega focuses on finite element simulation. PositionBasedDynamics is a set of tools to build a PBD solver. SPlisHSPlasH provides various solvers for fluid dynamics. Our work is mostly orthogonal to these at the moment.

3. Library Design

Library Organization Yocto/GL is comprised of a few small libraries. While we provide example applications, we consider the core libraries to be the part of Yocto/GL that are maintained. All libraries share common math types for short vector and matrices, and communicate mostly by passing arrays of basic types. Beside this common dependencies, most libraries are self-contained and do not depend on each other. In particular, we support the following libraries

1. *Yocto/Math*: fixed-size vectors, matrices, rigid frames, rays, bounding boxes and transforms
2. *Yocto/Random*: random number generation, Perlin noise, sampling and Monte Carlo integration utilities
3. *Yocto/Shape*: various utilities for manipulating triangle meshes, quads meshes and line sets, computation of normals and tangents, linear and Catmull-Clark subdivision, mesh loading and saving, procedural shapes generation, geometry utilities
4. *Yocto/Bvh*: two-level bounding volume hierarchy for fast ray intersection and closest point queries of triangle meshes, quads meshes, line sets and scene instances.
5. *Yocto/Image*: simple image data type, image resizing, filmic tonemapping, color correction, color grading, image loading and saving, procedural images, procedural environmental maps, color-space conversion
6. *Yocto/Trace*: path tracing of surfaces and hairs supporting area and environment illumination using a general multiple importance sampling scheme, microfacet materials, volume integration.
7. *Yocto/Scene*: scene data types (camera, textures, shapes, materials, instances) and evaluation of scene properties.
8. *Yocto/ModelIO*: low-level parsing and writing for Ply, Obj, Pbrt, and Yaml formats
9. *Yocto/SceneIO*: scene loading and saving of Obj, Pbrt, glTF, and a custom Yaml format

Code is organized by using two files per library, a header file for API definition and an implementation file. Only the math and random library use a single file since all functions are inlined for speed. We have investigated the use of a header-only deployment, but compilation times grew too much for it to be usable. Also, using header/implementation pairs helps hiding functionality that we do not want to make public to reduce dependencies.

Language Choice Yocto/GL is written in C++ for efficiency reasons and since most other libraries in graphics adopt that language. We also considered other modern languages, namely Go [Pik09], Rust [MKI14], C# and Swift . To test their viability, we ported the full path tracer, together with all needed supporting libraries. Go does not meet our needs mostly since it lacks operator overloading and good debugging support, and its garbage collector may be troublesome in large scenes. Rust code was harder to read since memory management is remarkably complex to understand. C# and Swift worked well in our port. C# code is easy to read, albeit more verbose, and common in graphics due to Unity. What concerned us was portability between operating systems and memory usage for large scenes. Swift simplifies the implementation since it augments C++ with transparent reference counting, simpler generics, error handling and faster compilation. Swift would have been our chosen language if it was not for portability. In the end, this port confirmed that for our needs C++ is still the best option. The main drawback of C++ with respect to these other languages is the lack of automated memory management. We mitigate this problem by using only value types which require no explicit memory management at all.

Procedural Style To quote John Carmack, id Software co-founder, “Sometimes, the elegant implementation is just a function. Not a method. Not a class. Not a framework. Just a function.” Whenever possible, Yocto/GL follows this guideline, using free functions and no objects. In Yocto/GL, most code handles only basic types, arrays of basic types or collections of them. Figure 2 shows examples of the procedural styles used in Yocto/GL. We use free functions since they are more composable than object’s methods as their APIs have explicit data passed to and return from them. Specialized types are used only when strictly necessary, for example to cache partial computation. An example of this API style is shown in Figure 2.

Data-Oriented Design When defining data structures, used rarely but necessary for example to describe whole scenes, we adopt a data-oriented design, rather than an object oriented one, which provides better readability, composability and performance. Figure 3 shows example of types defined in Yocto/Scene. In our design, data is explicit and directly accessible without encapsulation, and acted upon using free functions, instead of objects’ methods. In this manner, code is easy to understand and libraries can be extended by simply adding more functions without changing type definitions. We follow this guideline for all basic types, while for containers we match the STL design for consistency. Our data structures have a simple design, being defined as aggregates of arrays of value types. This makes data layout contiguous and memory access cache-friendly, improving efficiency. The main disadvantage of this design is the lack of encapsulation, which implies that APIs change if data representation does. For us, this is not a concern

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

```
// example of procedural API
vector<vec3f> compute_normals(const vector<vec3i>& triangles, const vector<vec3f>& positions);

pair<vector<vec4i>, vector<vec3f>> subdivide_catmullclark(
  const vector<vec4i>& quads, const vector<vec3f>& vert, int level, bool lock_boundary = false);

// example of using specialized types to cache computation
hash_grid make_hash_grid(const vector<vec3f>& positions, float cell_size);
void find_neighbors(const hash_grid& grid,  vector<int>& neighbors, const vec3f& position, float radius);
```

Figure 2: *Example of procedural coding style, for normal computation, subdivision surfaces and nearest-point queries. In Yocto/GL, we use specialized types, only when strictly necessary, preferring to model problems using functions rather than objects. This makes the API more composable.*

```
// textures store arrays of basic types
struct yocto_texture {
  string       uri = "";
  image<vec4f> hdr = {};
  image<vec4b> ldr = {};
};
// materials store basic types and references
struct yocto_material {
  string uri            = "";
  vec3f   emission      = {0, 0, 0};
  vec3f   diffuse       = {0, 0, 0};
  int     emission_tex = -1; // reference
  int     diffuse_tex  = -1; // reference
  // other material properties...
};
// instances store frames and references
struct yocto_instance {
  string  uri      = "";
  frame3f frame    = identity3x4f;
  int     shape    = -1; // reference
  int     material = -1; // reference
};
```

```
// shapes store arrays of basic types
struct yocto_shape {
  string        uri       = "";
  vector<int>   points    = {};
  vector<vec2i> lines     = {};
  vector<vec3i> triangles = {};
  vector<vec4i> quads     = {};
  vector<vec3f> positions = {};
  vector<vec3f> normals   = {};
  // other face-varying elements and vertex data...
};
// scenes store arrays of objects represented as values
// references between objects are specified as array indices
// we provide no constructors, destructors, assignments, etc.,
// since the default ones are well-defined for value types
struct yocto_scene {
  string                 uri         = "";
  vector<yocto_shape>    shapes      = {};
  vector<yocto_instance> instances   = {};
  vector<yocto_material> materials   = {};
  vector<yocto_texture>  textures    = {};
  // other object types...
};
```

Figure 3: *Example of scene data structure written in a data-oriented style. All data is explicit, freely accessing and defined as simple collection of basic types. All types have value semantic, meaning that assignment provides deep copies. We use integer references and avoid pointers altogether, to ensure maintain semantic and avoid memory bugs.*

since the basic underlying types are few and well accepted, e.g. it is unlikely that we will ever change how triangle meshes are stored.

Value Sematic Nearly all types in Yocto/GL have value semantic, meaning that assignments copy values rather than references. Most data is passed by value, or equivalently constant reference, to functions rather than modifiable reference. Value semantic helps reducing bugs since all state changes are explicit, instead of being hidden. Parallelization is also simpler for the same reasons. Using values is also quite helpful in writing code quickly, e.g. history-based undo systems can be implemented by just copying values in an array.

Value types are easy to write in C++ since value semantic is composable, meaning that if members of an aggregate type are value types, then the aggregate is a value type. Creating a new value type amounts to defining its member variables, without the need for constructors or destructors. This simplicity comes from the fact that C++ is a value-oriented language and has a value-oriented standard library.

We make exception to value semantic for types that wrap external resources, such as file handles or mutexes. In these cases, the wrapper resource needs to be acquired and released explicitly. Here we use C++ move semantic and disable copies. For the programmer perspective, this ensures proper semantic and resource acquisition and release, via C++ RAII, while the code reads like value semantic and requires no pointers.

References When object references are required, for example to store texture references in materials, we use integer indices that refer to positions in data arrays, rather than using C++ pointers directly. This is akin to use vertex indices for faces in indexed meshes. This is shown in Figure 3. There are several advantages for doing this. First, we avoid pointers altogether, thus reducing the chance of memory corruption bugs. Smart pointer would help here but at the price of speed and significant verbosity. Second, indices ensure value semantic when copying whole data structures, since copies are always deep rather than shallow. Finally, we observe speed improvements when switching from raw pointers to indices likely due to better cache utilization and memory fragmentation, since we allocate arrays of contiguous small values, rather than many small objects on the heap.

Extensibility Extending a library beyond what was initially intended for is a main concern for research projects. The main method to extend an object-oriented system is to add new types that conform to given interfaces, but this only extends functionality that

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

```
// light defined non-intrusively as external arrays
// of indices to geometric and environment lights
// together with sampling data associated with
// shapes and textures accessed by indices
struct trace_lights {
  vector<int>           instances       = {};
  vector<int>           environments    = {};
  vector<vector<float>> shape_cdfs      = {};
  vector<vector<float>> environment_cdfs = {};
};
// data is shared between data structures by using
// const views (span<const T>) that have equivalent
// read semantic to owning arrays (vector<T>)
struct bvh_shape {
  span<const int>   points    = {};
  span<const vec2i> lines     = {};
  span<const vec3i> triangles = {};
  span<const vec4i> quads     = {};
  span<const vec3f> positions = {};
  span<const float> radius    = {};
  vector<bvh_node>  nodes     = {};
};
struct bvh_scene {
  span<const bvh_instance> instances = {};
  vector<bvh_shape>        shapes    = {};
  vector<bvh_node>         nodes     = {};
};
```

Figure 4: *The data needed by the path tracer to sample lights is defined externally, without modifying the scene data structure. When index references cannot be used, safe access to raw data is provided by C++ view types, such as* span

was already planned for. Adding new functionality requires changing the types themselves, essentially forking the whole library. This is what our research group did for many years and, in our experience, this manner of extensibility does not work well when research projects are varied.

A procedural data-oriented design supports extensibility in a different manner. Adding new behaviors is quite easy since free functions can be added at will without changing existing code. Adding data to existing data structures can also be done easily using indices as references.

Let us consider the case of adding application specific data to the scene data structure, shown in Figure 3. The typical way to do this is to intrusively modify the scene types to include the additional data. This does not scale well as more applications are written since a type will contain data from all applications that might need it. Also, storing cached data on an object, as often done in object-oriented design, requires updating that data every time something changes, which makes writing interactive editors hard. In Yocto/GL, we store application specific data non-intrusively in external arrays indexed by reference indices. In this manner, data structures remain lightweight, while access to application data remains fast since it only requires array lookups. We make heavy use of this extensibility both within Yocto/GL libraries, where the scene is augmented with external data in this manner as shown in Figure 4, as well as in all our research projects that in this way do not require forking but at the same time do not bloat the whole codebase.

Views The only exception to the reference rules above is when raw data needs to be accessed by different libraries. An example, shown in Figure 4, is the implementation of a BVH that requires direct access to vertex positions. In this case, we use C++ memory view types, such as `span` and `string_view`, that are type safe and have APIs similar to owning types, such as `vector` and `string`. Similarly, when accessing external libraries we avoid memory copies altogether, preferring raw pointers instead, for example when interfacing with Embree [WWB*14]. Note though that all this code is hidden from the user and only present in the implementation, making Yocto/GL APIs memory safe.

Generic Programming The programming feature that required most refactoring was the use of generic programming through C++ templates. In general, there is little agreement in the community on how to handle this, with libraries that vary from a heavy use of templates, like Eigen [GJ*10], to no template code in the API, like Embree [WWB*14]. Yocto/GL favors the latter style for three reasons: readability, correctness, and compile times.

Let us consider the simple case of a low-level math library of short vectors and matrices, shown in Figure 5. One alternative is to make separate types for vectors of different dimension and element types, like `vec3f` or `vec2i`. Another alternative is to define a templated type that covers all cases, like `vec<T,N>`. The latter case feel simpler at first, but then forces all functions to be templated, making them less readable. Also, template type deduction does not work as well as specific types when used with C++ type inference, which means that types have to be specified more pedantically. Also, direct variable access requires full template specialization, which is a relatively advanced language feature.

In terms of correctness, we found that it is not easy to write templated code that works well when substituting all types. In a library like Yocto/GL which is used by students not trained in C++, this ended up being confusing. This confusion is not helped by the notoriously bad template error messages. In the end, we found that generic code was hard to write in a manner that was correct in all cases or at least gave clear indication as to when it would fail at compile time. Finally, even in a small library like Yocto/GL, compile times grew significantly as we move code from implementation files to header files, as required in the case of heavy template use. Most of our users did report that the increased compilation time was one of their main concerns. In the end, we settle for using little templates even if it feels wrong in C++ which is a language that prefers templates in its standard library.

4. Selected Library Features

In terms of graphics features, Yocto/GL has many similarity with other research libraries. Rather than simply listing all these features, we prefer to focus this section on describing design decisions that differ from others to highlight tradeoffs.

Simplicity Scales Yocto/GL focuses on code that is *simple, composable and scalable*. These three goals are strong constraints when it comes to choosing algorithms and data representations that may reduce the generality of the library or the efficiency of the approaches. In the end, throughout the design of Yocto/GL and in its current use, we always find that *simplicity scales* better to large environments, at least for our target applications. This is true for al-

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

```
// implementation without templates duplicates
// code but is readable for non experts
struct vec3f {
  float x = 0, y = 0, z = 0;
  vec3f(float x, float y, float z) { ... }
};
struct vec2f {
  float x = 0, y = 0;
  vec2f(float x, float y) { ... }
};
// operators are duplicate but straightforward
vec3f operator+(const vec3f& a, const vec3f& b) {
  return {a.x + b.x, a.y + b.y, a.z + b.z}; }
vec2f operator+(const vec2f& a, const vec2f& b) {
  return {a.x + b.x, a.y + b.y}; }
```

```
// implementation with templates is generic but complex
template<type T, int N>
struct vec {
  T _data[N];
  // simple constructors are slow compared to above
  vec(initializer_list<T> v) { /* more code here */ }
  // variadic constructors are hard to implement and use
  template<typename ... Args>
  vec(Args ... args) { /* more code here */ }
  // named members are unsafe or require specialization
  T& x() { /* more code here */ }
};
template<typename T, int N>
vec3f operator+(const vec<T,N>& a, const vec<T,N>& b) {
  /* use loops, but may go slow */
  /* use specialization/overloading, like non-templates */ }
```

Figure 5: *Comparison between using templates or nor in the simple case of short vectors.* Right: *Using no templates requires code duplication, but the code is straightforward for all users.* Left: *Templates support well generic code but are harder to read for non experts and more cumbersome to ensure efficiency in all cases.*

gorithms, but even more for data structures whose complexity often makes them significantly harder to optimize well.

Specialization As for many software projects, the main concern is that increasing functionality scales poorly with simplicity, robustness, and maintainability. In our domain, we found that the best tradeoff is to use a small set of data primitives and graphics algorithms, which are general and fast, albeit not necessarily optimally efficient. In a way, we follow the Unix philosophy, that in the words of Doug McIlroy is "Write programs that do one thing and do it well".

Physical Units and Transforms Throughout Yocto/GL we assume that values are specified in physical units. This obviously matters in rendering and simulation since these algorithms compute physical quantities. In modeling, this was not a concern until recently, when the advent of photogrammetry and 3d printing made a direct connection with real-world objects.

A corollary of this is that transforms in Yocto/GL are represented as rigid frames, represented as a 3-by-3 rotation matrix plus an origin position, rather than generic 4-by-4 matrices. Using rigid frames means that physical quantities like energy or mass are not altered in instances. Furthermore, photogrammetry textures are also not scaled.

Memory Scalability In terms of scalability, the largest concern we focused on was reducing memory consumption, since most Yocto/GL users have laptops. What surprised us is that a value-based design using indices as references scales better than pointer-based object-oriented ones. To illustrate our point, let us consider the simple case of representing instances and compare Yocto/GL to a typical graphics library. In Yocto/GL, instances store a frame and two indices to a shape and a material, taking 56 bytes per instance. Instances are allocated as values, so arrays of instances are laid out in contiguous memory. In a typical object-oriented setup, with heap-allocated objects referenced with raw pointers and matrices used as transforms, the same instance takes roughly twice the memory with many small objects allocated on the heap. Using smart pointers further increases memory usage in a platform dependent way.

Scene Format Parsing Yocto/GL supports loading and saving for many scene formats, namely Obj, Pbrt, Ply, glTF and a custom Yaml format. We provide both low-level parsers as well as loading and saving of internal scene data structure. For all these parsers, there are well-established open-source solutions that we tested. In the end though, we found that parsing large scene files was cumbersome with current libraries and developed our own methods. The main limitation of current low-level parsers is that they keep a copy of the whole scene in memory. For large scenes this did not scale for our datasets. For this reasons, we designed SAX-like low-level parsers and writers for Obj, Pbrt, Ply and Yaml and use those in our work. For each formats, our parsers and writer processes one command at a time and are designed to give the application full control on data access, memory allocation and control flow. Surprisingly, this led to simpler higher level scene loading and saving since there was no need to explicitly convert between data structures, but was instead sufficient to just execute the command stored in each format.

Figure 6: *Example of computational geometry feature. A geodesic distance field, whose isolines are visualized as red stripes, is computed with the graph solver provided by Yocto/GL. The mesh in the example has 7.2 million triangles and the geodesic distance field was computed in 0.458 seconds on a laptop. Model courtesy of Stanford Graphics Lab [Sta94].*

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

Figure 7: *Example of geometry and material types. Geometries include quad and triangle meshes, displaced subdivision surfaces and hair. Materials include matte, plastic, metal, volumetric glass. Models courtesy of Stanford Graphics Lab [Sta94] and Blender.*

Geometry We support only four kinds of geometric primitives: points, lines, triangles and quads, shown in Figure 7. Curves and subdivision surfaces are supported in part but require explicit tesselation for most functionality. While this approach is common in computational geometry library, it is less so in rendering ones. For example, Intel's Embree [WWB*14] has a design similar to ours, albeit it natively supports curves and subdivision surfaces, while Pbrt [PJH16] uses an object-oriented design to support natively many different shape types.

Just like Libigl [JP17], we do not use mesh data structures, but rely only on an indexed mesh representation. Adjacencies are computed when required. This naturally supports triangle and quad meshes, line and point sets. Where we differ more from other libraries is that we support face-varying data, represented by optionally storing a different topology for each vertex property. The use of face-varying data is particularly helpful in animation and mesh processing where vertex duplication causes concerns.

When it comes to geometry algorithms, we target approaches that run on large meshes and do not require complex data structures. As an example, our implementation of subdivision surface is based on [WS04], which is an averaging scheme that only requires a standard dictionary to store edges. Another example, shown in Figure 6, is our graph-based geodesic solver that use a graph that is only accurate for high tesselation, but can be made quite efficient in that case.

Materials Just as shapes, we represent materials using one "shader" that can represent a variety of materials including plastic-like substrates, reflective metals, transmissive dielectrics and homogeneous volumes, for volumetric effects and subsurface scattering. Example materials are shown in Figure 7. This follow a trend in industry of using similarly general shaders, such as Disney's unified model [Bur15], Pixar's layered model [HL17] and Autodesk's standard surface [GPA*16]. Again, here we differ from today's research renderer Pbrt [PJH16] and Mitsuba [Jak10] that use an object-oriented design with many special cases.

Rendering In Yocto/GL all rendering is performed with a progressive path tracer suitable for both offline and interactive rendering. We use a unidirectional path tracer that supports our material model using multiple importance sampling throughout the renderer to reduce noise. Homogeneous volumes and subsurface scattering are implemented as random walks with MIS. Our implementation though takes a sharp departure from prior work and avoid shadow rays and area integration, preferring to formulate the entire renderer in the angular domain. Note that this does not prevent MIS but strengthens it, since hard cases such as "shadows" through glass and direct illumination inside volumes are handled without special cases. Figure 1 and Figure 8 show images generated with our path tracer, while Figure 9 show a few frames of an editing session using our interactive path tracer.

A unidirectional renderer is certainly not state-of-the-art, since many other methods are known, such as bi-directional and metropolis methods. Research renderer, such as Pbrt and Mitsuba, implement many of these algorithms. Our findings though is that these algorithms may help in some specific cases, but they are not particularly helpful for common scenes. This is the same conclusion reached by the production rendering industry [FHP*18] where all current systems are unidirectional path tracers, including Weta's Manuka [FHL*18], Pixar's Renderman [CFS*18], Autodesk's Arnold [GIF*18] and Disney's Hyperion [BAC*18].

One unexpected outcome of our focus on interactive path tracing is that we now mostly use a simple CPU raytracer instead of a GPU rasterizer for interactive rendering. Beside code complexity, the main limitation of GPU viewers is that they do not scale as well to large scenes due to VRAM constraints, especially on mobile GPUs found on laptops. This is the reason why we removed all GPU code from the main library and focused on CPU-only solutions, even for interactive applications.

Example Project As example project built on top of Yocto/GL we present SceneGit [CP19], a version control system for 3D scenes comprised of shapes, materials, textures and animations which is able to detect changes between edited versions of scenes. When two users edit concurrently the same scene, the system can identify the differences and merge the two versions automatically, also detecting editing conflicts if any. Objects are versioned at their finest granularity, in order to make repositories smaller and minimize the chance of merge conflicts. The system is robust, efficient both in terms of space and time, and scales to large scenes, as showed in the Figure 10.

5. Conclusions

This work presents the design rationale of the Yocto/GL library. In conclusions, Yocto/GL differs from other libraries since it takes a less common tradeoff in software design, focusing on code that is short, simple, composable and scales with scene complexity. The price we pay is that we support only a limited feature set, compared to other libraries, and avoid the use of GPUs. In the future, we plan to extend the library while maintaining its design strengths. In particular, we plan to increase the computational geometry aspects by integrating code directly from Libigl, providing wrapper support for sparse linear solvers, augment the renderer with non-homogenous volumes and learned importance functions, and integrate a particle-based physical solver. In doing so, we will still follow the main design tradeoffs listed before, which are and will remain possible limitations.

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

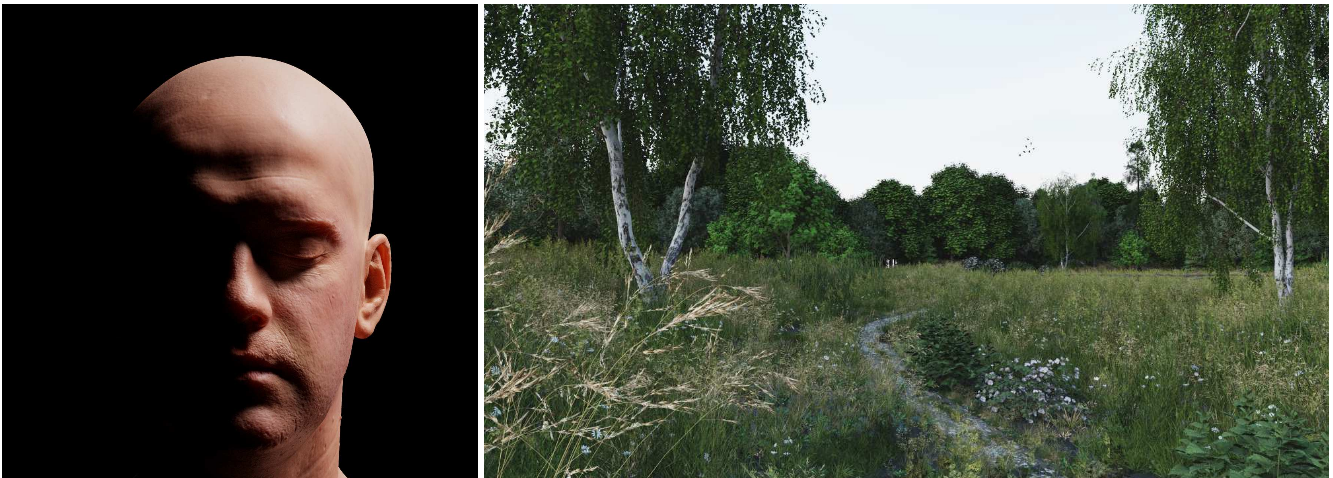

Figure 8: *Example of realistic renderings made with Yocto/GL path tracer.* Left: *Rendering of human skin, simulated as a participating medium with volume integration. Model courtesy of I-R Entertainment Ltd. [McG17].* Right: *Rendering of a complex scene that makes heavy use of instancing. Model courtesy of Jan-Walter Schliep, Burak Kahraman, and Timm Dapper [JH04].*

Figure 9: *Example of interactive rendering and editing of large scene.* (Left:) *Our progressive path tracer gives immediately a low-sampled preview to allow real-time interaction. The image progressively converges as more samples are computed.* (Middle): *The user can move the camera to interactively explore the environment.* (Right) *The scene can be edited in real-time, by changing materials and moving shapes. Model courtesy of Amazon Lumberyard [Lum17, McG17].*

Acknowledgments

This work was partially supported by Intel Corporation and MIUR under grant PRIN *DSurf* and *Dipartimenti di eccellenza 2018-2022* of the Department of Computer Science at Sapienza.

References

[BAC*18] BURLEY B., ADLER D., CHIANG M. J.-Y., DRISKILL H., HABEL R., KELLY P., KUTZ P., LI Y. K., TEECE D.: The design and evolution of disney's hyperion renderer. *ACM Transactions on Graphics 37*, 3 (2018). 7

[Ben18a] BENDER J.: Positionbaseddynamics. https://github.com/InteractiveComputerGraphics/PositionBasedDynamics, 2018. 3

[Ben18b] BENDER J.: Splishsplash. https://github.com/InteractiveComputerGraphics/SPlisHSPlasH, 2018. 3

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

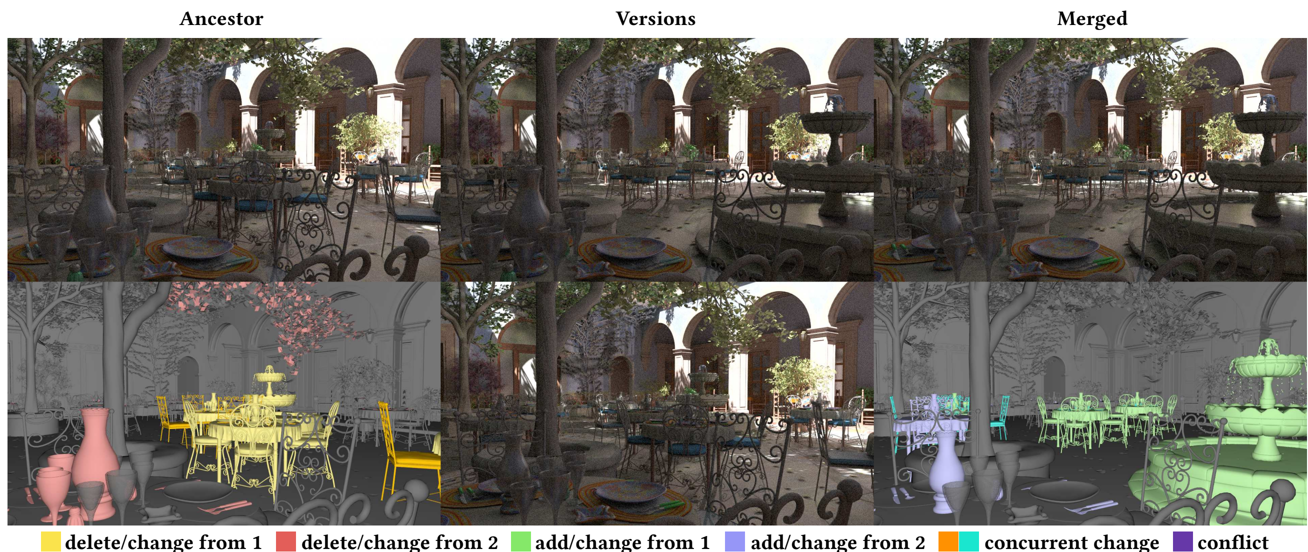

Figure 10: *Example of project built on top of Yocto/GL: A version control system for 3D scenes. Two scene (*middle top and middle bottom*) are created concurrently by editing a common ancestor (*top left*). The system detects changes between each version and the ancestor, then merges the changes automatically (*top right*). The highlight colors show added and changed elements on the merged scene (*bottom right*), and deleted and changed elements on the ancestor (*bottom left*). The color of the edits, shown in the legend below the figure, indicates their provenance: green/yellow for version one, blue/red for version two, cyan/orange if the same element was edited in both versions, purple for conflicts. Model courtesy of Guillermo M. Leal Llaguno [McG17].*

[Bur15] BURLEY B.: Extending the disney brdf to a bsdf with integrated subsurface scattering. In *ACM SIGGRAPH 2015 Course Physically Based Shading in Theory and Practice* (2015). 7

[CCC*08] CIGNONI P., CALLIERI M., CORSINI M., DELLEPIANE M., GANOVELLI F., RANZUGLIA G.: Meshlab: an open-source mesh processing tool. In *Eurographics Italian chapter conference* (2008), vol. 2008, pp. 129–136. 2, 3

[CFS*18] CHRISTENSEN P., FONG J., SHADE J., WOOTEN W., SCHUBERT B., KENSLER A., FRIEDMAN S., KILPATRICK C., RAMSHAW C., BANNISTER M., ET AL.: Renderman: An advanced path-tracing architecture for movie rendering. *ACM Transactions on Graphics 37*, 3 (2018). 7

[Cou10] COUMANS E.: Bullet physics engine. *Open Source Software: http://bulletphysics. org 1*, 3 (2010), 84. 3

[CP19] CARRA E., PELLACINI F.: Scenegit: A practical system for diffing and merging 3d environments. *ACM Trans. Graph. 38*, 6 (2019). 7

[FHL*18] FASCIONE L., HANIKA J., LEONE M., DROSKE M., SCHWARZHAUPT J., DAVIDOVIČ T., WEIDLICH A., MENG J.: Manuka: A batch-shading architecture for spectral path tracing in movie production. *ACM Transactions on Graphics 37*, 3 (2018). 7

[FHP*18] FASCIONE L., HANIKA J., PIEKÉ R., VILLEMIN R., HERY C., GAMITO M., EMROSE L., MAZZONE A.: Path tracing in production. In *ACM SIGGRAPH 2018 Courses* (2018), ACM, p. 15. 7

[GIF*18] GEORGIEV I., IZE T., FARNSWORTH M., MONTOYA-VOZMEDIANO R., KING A., LOMMEL B. V., JIMENEZ A., ANSON O., OGAKI S., JOHNSTON E., ET AL.: Arnold: A brute-force production path tracer. *ACM Transactions on Graphics 37*, 3 (2018). 7

[GJ*10] GUENNEBAUD G., JACOB B., ET AL.: Eigen v3. http://eigen.tuxfamily.org, 2010. 5

[GPA*16] GEORGIEV I., PORTSMOUTH J., ANDERSSON Z., HERUBELAND A., KING A., OGAKI S., SERVANT F.: Autodesk standard surface. https://autodesk.github.io/standard-surface, 2016. 7

[HL17] HERY C., LING J.: Pixar's foundation for materials: Pxrsurface and pxrmarschnerhair. In *ACM SIGGRAPH 2017 Course Physically Based Shading in Theory and Practice* (2017). 7

[Jak10] JAKOB W.: Mitsuba renderer, 2010. http://www.mitsuba-renderer.org. 2, 7

[JH04] JAKOB P., HUMPHREYS: Scenes for pbrtv3, 2004. URL: https://pbrt.org/scenes-v3.html. 8

[JP17] JACOBSON A., PANOZZO D.: Libigl: Prototyping geometry processing research in c++. In *SIGGRAPH Asia 2017 Courses* (2017), pp. 11:1–11:172. 2, 7

[Liv19] LIVESU M.: Cinolib: a generic programming header only c++ library for processing polygonal and polyhedral meshes. *Transactions on Computational Science XXXIV* (2019). https://github.com/mlivesu/cinolib/. 3

[Lum17] LUMBERYARD A.: Amazon lumberyard bistro, open research content archive (orca), July 2017. http://developer.nvidia.com/orca/amazon-lumberyard-bistro. URL: http://developer.nvidia.com/orca/amazon-lumberyard-bistro. 8

[McG17] MCGUIRE M.: Computer graphics archive, 2017. URL: https://casual-effects.com/data. 8, 9

[MKI14] MATSAKIS N. D., KLOCK II F. S.: The rust language. In *ACM SIGAda Ada Letters* (2014), vol. 34, ACM, pp. 103–104. 3

[Pik09] PIKE R.: The go programming language. *Talk given at Google's Tech Talks* (2009). 3

[PJH16] PHARR M., JAKOB W., HUMPHREYS G.: *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016. 2, 7

[SSB13] SIN F. S., SCHROEDER D., BARBIČ J.: Vega: non-linear fem deformable object simulator. In *Computer Graphics Forum* (2013), vol. 32, Wiley Online Library, pp. 36–48. 3

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

[Sta94] STANFORD U.: The stanford 3d scanning repository, 1994. URL: http://www.graphics.stanford.edu/data/3Dscanrep/. 6, 7

[Stu18] STUDIOS W. D. A.: Moana island scene, 2018. URL: https://www.technology.disneyanimation.com/islandscene. 1

[The19] THE CGAL PROJECT: *CGAL User and Reference Manual*. CGAL Editorial Board, 2019. URL: https://doc.cgal.org/4.14/Manual/packages.html. 2

[WS04] WARREN J., SCHAEFER S.: A factored approach to subdivision surfaces. *IEEE Comput. Graph. Appl. 24*, 3 (2004), 74–81. 7

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: A kernel framework for efficient cpu ray tracing. *ACM Trans. Graph. 33*, 4 (2014). 5, 7

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.