Art and Technology at Pixar, from Toy Story to today

SIGGRAPH ASIA 2015 Course Notes

Course Organizer

Ryusuke Villemin
Pixar Animation Studios

Presenters

Christophe Hery
Pixar Animation Studios

Sonoko Konishi
Pixar Animation Studios

Takahito Tejima
Pixar Animation Studios

Ryusuke Villemin
Pixar Animation Studios

David G Yu
Pixar Animation Studios

Course Description

Technology has always played an important part in Pixar's movie making process, starting with Toy Story over 20 years ago. This course will take you on a journey through that technical evolution, focusing on how story drove our technical designs and how we utilized new technical advancements to tell more appealing stories. Finally we'll focus on several fundamental changes that happened recently. Prman's long used REYES algorithm, has been replaced by RIS, a modern raytracing engine. At the same time, the shading language used for more than 20 years, RSL, was replaced by C++ and OSL shaders. Pixar's famous subdivision surface algorithm was made open-source, in order to make it available across 3D software, and promote its development at a larger scale.

Level of Difficulty: Intermediate

Intended Audience

Practitioners from CG Animation, VFX Fields and videogames, plus researchers interested in the current trend in the industry.

Prerequisites

A basic understanding of computer graphics, modeling, lighting and shading models.

Course Website

`http://graphics.pixar.com/Library`

Contact

`rvillemin@pixar.com`

About the Presenters

CHRISTOPHE HERY joined Pixar in June 2010, where he holds the position of Senior Scientist. He wrote new lighting models and rendering methods for Monsters University and The Blue Umbrella, and continues to spearhead research in the rendering arena. An alumnus of Industrial Light & Magic, Christophe previously served as a research and development lead, supporting the facility's shaders and providing rendering guidance. He was first hired by ILM in 1993 as a senior technical director. During his career at ILM, he received two Technical Achievement Awards from the Academy of Motion Pictures Arts and Sciences.

SONOKO KONISHI joined Pixar in 1994 as a Jr. TD after graduating from a fine arts college in the States. She worked on Toy Story, Toy Story 2 and Toy Story 3 in various departments such as lighting, modeling and character rigging. Over the past 20 years Sonoko has predominantly focused on feature film production but she has also worked on the production of shorts, TV Specials and promotional materials. She currently works for the simulation & digital tailoring department.

TAKAHITO TEJIMA is a senior software engineer at Pixar Animation Studios, focusing on real-time GPU rendering. He previously worked at Polyphony Digital Inc. in Japan, as a lead developer of console video game Gran Turismo series for 12 years. At Pixar, he is currently working on developing proprietary animation software and also OpenSubdiv, the open source subdivision surface library.

RYUSUKE VILLEMIN began his career at BUF Compagnie in 2001, where he co-developed BUF's in-house raytracing renderer. He later moved to Japan at Square-Enix as a rendering lead to develop a full package of physically based shaders and lights for mental ray. After working freelance for Japanese studios like OLM Digital and Polygon Pictures, he joined Pixar in 2011 as a TD. He currently works in the Research Rendering department.

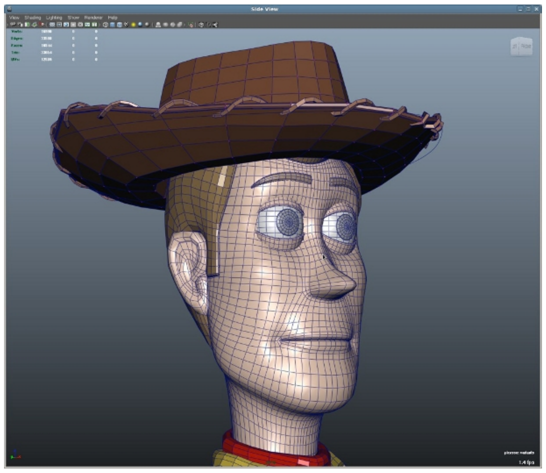
DAVID G YU has been at Pixar since 2000 working on the Presto animation system (and its predecessor Marionette) with a focus on advancing the use of GPU technology. Prior to that David was at Silicon Graphics Inc (SGI) developing workstation hardware and software for OpenGL and visual simulation.

Presentation Schedule

08:30-08:35	**Introduction** *(Villemin)*
08:35-09:40	**20 years of ToyStory** *(Konishi)*
09:40-10:15	**Hydra Real-time Render Engine** *(Yu)*
10:15-10:30	**Break**
10:30-11:05	**OpenSubdiv 3.0 introduction** *(Tejima)*
11:05-12:05	**Prman RIS** *(Hery and Villemin)*
12:05-12:15	**Q&A**

20 years of ToyStory

by Sonoko Konishi


(a) Toy Story 3 Woody

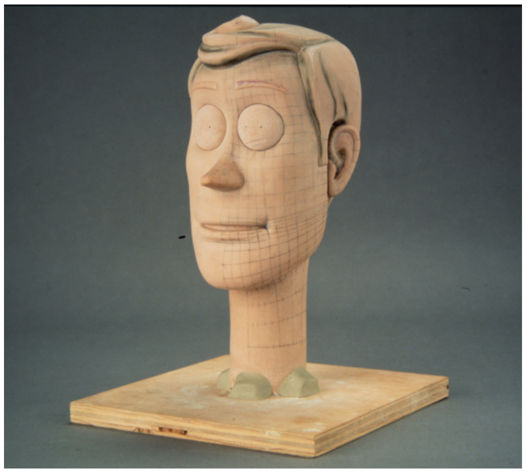

(b) Toy Story Woody Maquette

Figure 1: ©Disney/Pixar 2015

1 Storytelling with technology

In 1995 Toy Story became the first feature-length film created using computer animation but beyond the novelty of CG animation was something deeper, a highly polished script that worked on many levels, breathing fresh air into the world of animated features. From Andys room in Toy Story to the inside of Rileys head in Inside Out, from current time Andy heading to college in Toy Story 3 to post-apocalyptic future in Wall-E the story drives our technology, looking to new techniques and tools to enrich the story but never allowing those tools to drive our story decisions. So, how did we get there?

Our focus on story extends beyond the traditional development practices of film screenings and reviews with our development partner Disney. From very early on Ed Catmull and John Lasseter built an environment where employees feel safe to bring new ideas and everybody can communicate freely. As we work on the films, the entire crew is encouraged to attend work in progress screenings and give notes on the film. We started this sense of creative collaboration when we were a mere handful of artists & technicians and it lives on today in a team that is over 1200 people. This approach exists beyond story development. Within the technical departments, we treat our tools and processes the same way we approach story telling, refining and polishing through collaboration among artists, always looking for feedback about how to bring the highest quality we can achieve for our films. Always asking ourselves, how can we inspire the story.

John Lasseter says The art challenges technology. Technology inspires the art. This is a core tenet of our studio; one inspired by the early days of Disney. Snow White, for example, represents a tent pole of cinematic & technology advancement combining realistic movement and emotion in the human characters with caricatured dwarfs and forest animals. They also invented the multi-plane camera which allowed artistic direction and cinematography to create a more believable world through the three-dimensional depth and movement within the frame. It is an important lesson, showing that technology is not just a cold tool but an important part of our creative language.

An example of this, is how we treated camera movement in Toy Story. We tried very hard to treat our virtual cameras as if they were real. Following traditional cutting patterns and treating the camera movement as if it had weight and form. Always asking ourselves, how does this choice advance the story? But we had limitations, depth of field was extremely expensive and so it was restricted to only the shots that were significantly enhanced by it. On Toy Story 2, our ability to move the camera was more sophisticated; as artists we had improved and our tools had been developed to allow softer lighting and depth of field allowing us to tell the story in a more cinematic way. With Toy Story 3, we placed a greater emphasis on drawing the audience into the toys prospective of the human world; using rack focus to the guide the audience eyes, character POVs to tell their emotions and richer color nuances brought by point-based global illumination.

The development of these techniques is facilitated by modern production processes. Creative thought needs to be uninterrupted and ideas and progress need to be communicated in timely manner to keep the creative flow going. But during Toy Story, we still had to resort to analog processes. Story boards were drawn on paper and scanned in one at a time. The editor would then have to go frame by frame to recreate the timing of the original pitch. Layout artist and animators could check their sequences and animatics with playblast but in order for the shots to be delivered to editorial, we would transfer them to 3/4 inch video, editorial would then digitize the shots and cut them into the reel. For A Bugs Life we developed a digital delivery system allowing our editors to spend more time on the creative aspects of editing and less time on dealing with importing shots. We are constantly tweaking our database systems, asset control, shot deliveries and renders to cater to the production needs of 300 crews.

As our technology for creating films developed, so did the manner in which audiences experience them in the theatre. For Toy story each frame was projected through a set of RGB filters and output to film and development. Our production schedule had to respect reel-delivery and it was a moment of celebration as each reel was physically carried out of the studio. By 1999 a limited number of theaters had been equipped with Digital Light Processing (DLP) cinema and Toy Story 2 was shown to audiences with true digital to digital quality experience. In 2010, Toy Story 3 entered world of stereo projection allowing the audience to truly feel the scale of the toys world, something that was especially felt in the scale of the furnace in the final scene. In our latest installment, Inside Out, high dramatic range (HDR) laser projection to experience a richer range of light and color.

2 Animating with technology

The animation tools used for Toy Story were not created over night. Pixar had been developing the Menv (Modeling Environment) animation system for nine years and John Lasseter was a big part of of the UI development. Many of the fundamental design concepts developed by John Lasseter still exist in our tools today.

A common question today is Why do you still work in a proprietary platform? It was understandable for Toy Story because the tools didnt exist but Why now? Proprietary tools make it easier to maintain a custom look and feel for our assets; we can automate the upgrade process and maintain older versions of our assets for study and education. If you look at Toy Story 3, 58 animators worked on that film, but only 17 of them worked on Toy Story 2 and only 4 of them also worked on the original Toy Story. Yet the animation and characters have a consistent look and feel across all of the films. This is because we could always look back to the older assets, how the moved, behaved and animated.

During that time, Toy Story characters went through three major upgrades just for their surface construction alone. For Toy Story, the surfaces were created with stitched nurbs surfaces. During Geris Game we created techniques for animating and rendering subdivision surfaces and the original cast of Toy Story was remodeled for Toy Story 2 using subdivision surfaces, allowing us to achieve a wider range of complex facial expressions. We also pushed the appearance of human models with new skin and hair shaders that, while still a caricatured version of reality, contained more detail and sophistication. Eleven years after Toy Story 2 we embarked on Toy Story 3. We still wanted to maintain the look and feel of the original characters but they had to fit into a more current time. We added additional wear and tear to the toy characters for all those years past. Andy is now a grown up and needed a new facial rig, including hair that is both rigged and simulated. We also introduced subsurface scattering into the hair and skin shaders to bring a more translucent appearance. With Brave in 2012, Menv got a major update and it also meant updating our heroes from Toy Story to meet the requirements of the new Universal Scene Description (USD).

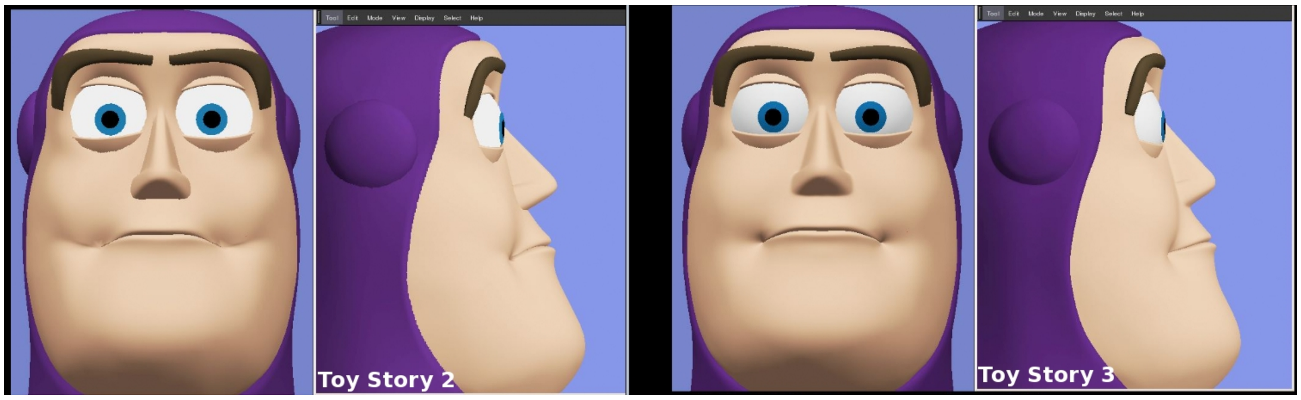


Figure 2: Buzz Lightyear ©Disney/Pixar 2015

All of these revisions come with increased complexity (Fig 2). Twenty year ago Woody had only 596 controls. By Toy Story 3 he had 2245. As our technology progress and animators skills became more and more refined we keep adding more and tools for them to control and shape the characters. But we cant forget what was achieved with those early rigs.

Toy Story 3 animation supervisor recalls how the first 2 films of Toy Story series had less animation controls and yet still express amazing animation with a small number of controls. In Toy Story, we had limited desktop computer performance, so many of characters had very simple geometry standins. Animators were using simple cylinders and cubes to block in the animation and turned on the high resolution surface called PAT for refining the performance. At that time turning on the final mesh slowed the machine to crawl so animators learned how to work around it by concentrating on the body language first and only switching to the detail mesh for finishing. It is this focus on acting in broad clear gestures that defines our characters and while we add a lot of controls for our animators, we are always looking for ways to automate animation behaviors such as foot ground contact and tire deformation on race tracks so that animators can focus on acting.

But kinematic rigs are not the only method to achieve convincing animation. Starting after Toy Story 2 and continuing with Toy Story 3 & Brave we focused on developing simulation tools for cloth and hair. However, the ability to run complex simulations is not sufficient; the movement of hair and cloth directly influences the audiences perception of a character and to integrate simulation into the story process requires direct-able simulation.

3 Enrichment with technology

3.1 Modeling

During Toy Story many characters started with a clay sculpt. We would then mark out the control vertices on the sculpt and digitize them with a Polhemus, a pen-like input device which we would touch to each point on the surface and click to register each point within the computer (Fig 1). It was a painstaking process that could not be interrupted or you would have to start over again. Today, we 3D scan the main character maquette or model in 3D directly. The scan provides us a good sense of overall proportion and volume which we use as a guide as we iterate with the director and adjust the topology for rigging. Using these processes we were able to create over 300 animatable characters for Toy Story 3 compared to the 76 that we were able to do for the original Toy Story in shorter time.

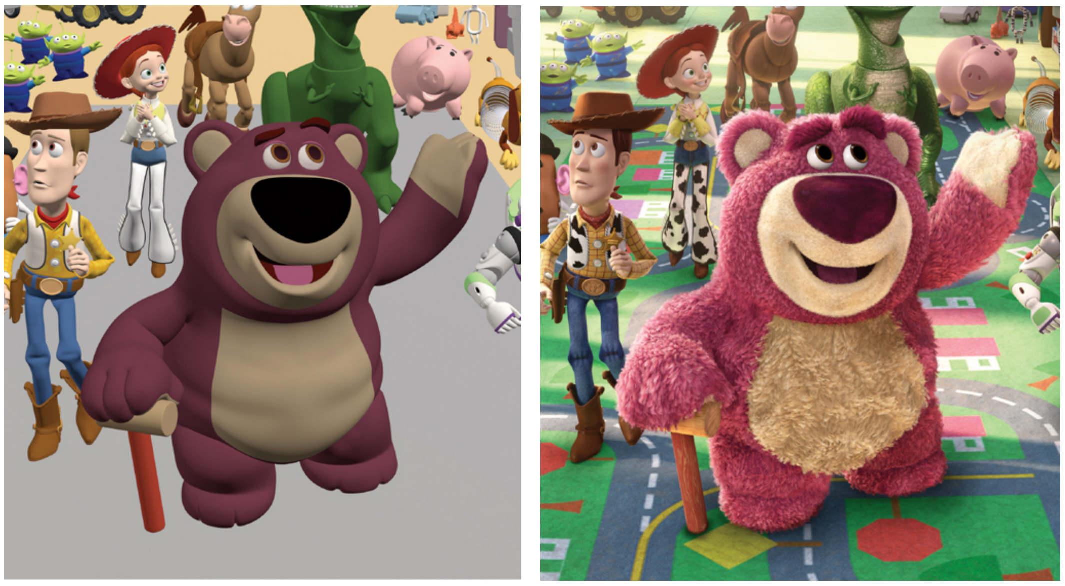

(a) Rigged wrinkles on Lotso (b) Final render with fur

Figure 3: ©Disney/Pixar 2015

Lots-o-Huggin Bear, the pink plush bear was a good example of where technology pushed the art. Without global illumination his fur would not have looked believable. As a matter of fact, there is one shot in the original Toy Story where Lotso made his first appearance. We also had to develop new rigging techniques to support a character with no internal bone structure. This included rigging the wrinkles so that animation would have precise control over how they where emphasized (Fig 3).

For Toy Story 3, we placed a lot of emphasis on improving our cloth pipeline so that it would be accessible to more modelers. Before Up, creating clothing followed a 2D pattern tailoring system which limited the available staff to people with drafting skills. In Toy Story 3, a 3D cloth system was in place and the modeler can model garment much like building a standing clothes and turn them into simulate-able cloth. The goal was that more modelers can build garments without deep understanding of pattern drafting knowledge.

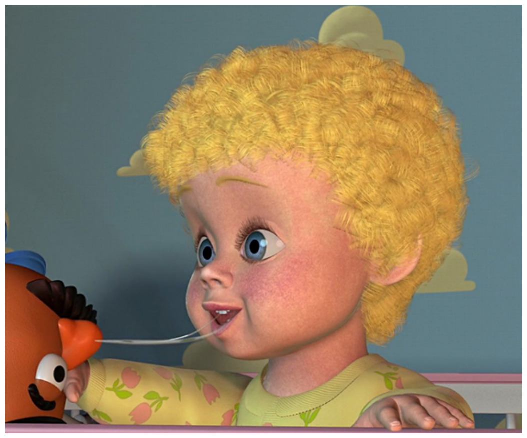
A member of Lotsos gang, Big Baby, the life like baby doll with a plastic head and limbs but with soft stuffed body benefited a lot from this new pipeline. His face and limbs were rigged much like the rest of the characters, however to create his stuffed bodice, the roughly rigged body core was wrapped in cloth and ran simulation to cover animated body. John Lasseter always remind us that stay true to the material and Big Babys appeal and articulations were achieved by hybrid of kinematic rigging and cloth simulation.

3.2 Shading

The shading of our characters presents a unique design challenge. We always strive to improve our shading models and level of detail, but at the same time, at a glance the classic characters need to feel like they came from the same family. Woodys costume for instance, as the Toy Story series progress, his denim jeans and plaid shirt have received weave pattern, sewing seams and subtle wear as year passes. It is this constant improvement while maintaining original feel that creates a sense of timelessness with our characters and add characters their story.

3.3 Simulation

Toy Story and Toy Story 2 did not have character effects that have become a standard part of todays film production. An example of something that appears to be dynamic is Slinkys spring body but even that was a rigged system with procedural follow through. For humans, the garments and hair was modeled and rigged as a part of the character mesh. Even Buster, whose hair was groomed using a prototype hair grooming tool developed for Monsters Inc was animated with kinematic hair. But Monsters inc allowed us to develop new tools for dynamic simulation. Initially put in place for Sullivans fur and Boos t-shirt these tools soon became a standard part of our prices called character effects. Toy Story 3 was no exception and we took advantages of these tools to refine our characters further using fully simulated cloth and hair.


(a) Toy Story Molly Hair

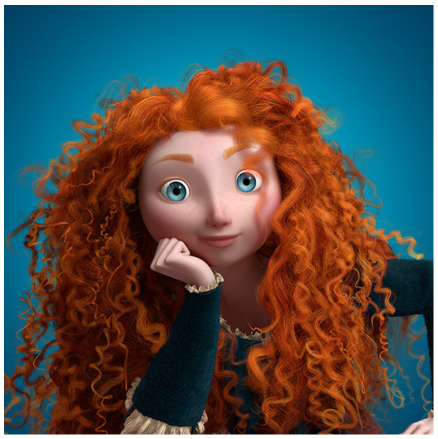

(b) Brave Merida Hair

Figure 4: ©Disney/Pixar 2015

After Toy Story 2, we were aiming for more ambitious cloth and hair work (Fig 4). As exemplified in many Disney Animations the shapes and behavior of cloth and hair are as expressive as their character and emotions. And Brave was the perfect story to apply these new approaches. Meridas fiery personality couldnt have been expressed fully without the technical development of curly hair and the Scottish costumes woundt be authentic without accurate multi layered cloth simulation.

4 People with technology

Throughout production we treat our success and mistakes as part of the same learning process. After each department wraps we have a post-mortem, evaluating the currently active technologies and processes looking for places to improve and evolve. As a result of these reviews some technologies will be retired or rewritten others live on. Sometimes these changes will be big, for example, 20 years after the original Toy Story, Menv was replaced with our new animation system Presto for Brave and currently incorporating lighting and shading workflow with Katana. Since a film runs for over 2 years, we have to be very cautious about which tools and technology we commit to producing a film.

From the earliest days of Pixar and the industry, we have always looked to experts from different disciplines to experiment and adopt new technologies. A practice we continue today; working with academic community and employees with research backgrounds while we open our studios to more young people through internship and residence program. Under the Disney umbrella, now we collaborate with Disney, Disney Research and I.L.M. and keep inspired by each other.

When Toy Story was released, Steve Jobs said it was our 'Snow White'. That we had married technology and story to create characters that people fall in love with. It is this balance that of art and technology that we strive to develop and not only is Toy Story a part of people's lives but the technologies that we created have allowed artists from all over the world to tell their own stories.

Hydra Real-time Render Engine

by David G Yu

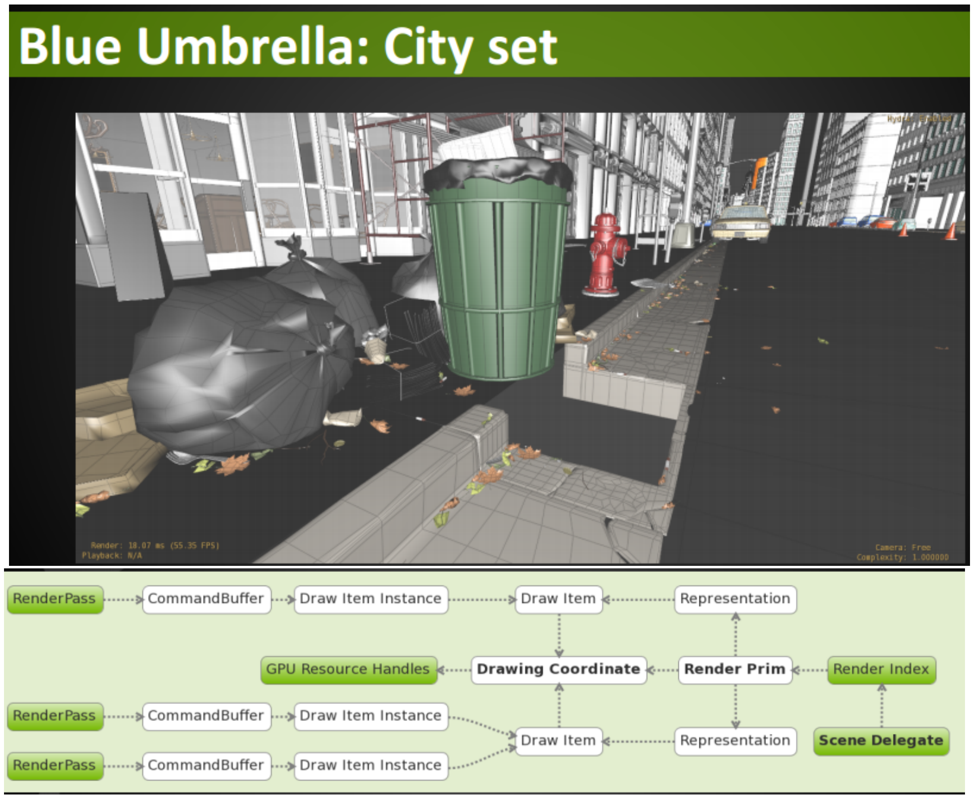


Figure 1: The Blue Umbrella©Disney/Pixar 2015

Abstract

Pixar's Hydra is a real-time render engine designed for the assets and workflows used in the production of animated feature films. Hydra delivers high performance with the scalability to support interactive animation of fully rigged characters as well as the pre-cached data sets used for large crowds, detailed set dressing, and dynamic special effects. The implementation takes full advantage of the latest GPU techniques including multi-threaded data preparation, GPU compute, and instanced, indirect drawing. Hydra incorporates OpenSubdiv and an extensible shading system to support the high-fidelity needed to allow production artists to work with confidence and in context.

Topics include:

- architecture details
- OpenSubdiv and Universal Scene Description (USD) integration
- GPU parallel instancing and frustum culling
- use in Presto and 3rd party DCC tools

1 Introduction

Hydra is the real-time render engine used within Pixar to display production assets. Originally developed as part of the Universal Scene Description project, it has become a foundation component for continued development of high-performance and high-fidelity rendering within the Studio.

Scalability and flexibility are paramount since real world film production data is typically optimized for more traditional off-line rendering and not designed with consideration of real-time constraints.

2 Architecture

One of the key realizations is that the scenegraph and render engine inputs are correlated, but distinct, and the final input into drawing hardware is an extremely simple stream of commands. Additionally, we acknowledge that at the heart of hardware rendering lies a caching problem, in which GPU resources must be tracked back to the source scenegraph for invalidation.

We introduce the notion of a RenderIndex which holds light-weight Render Prims, which in turn fetch their data from a client scenegraph via a Client Scene Delegate. Render Prims also hold references to GPU resources, allocated by a Resource Registry. In this way, the RenderIndex and Render Prims form a map from the client scenegraph to the required GPU resources and can be compiled down to a command stream.

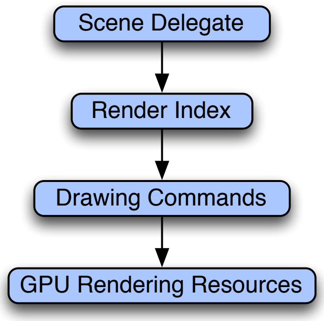


Figure 2: Architecture Overview

3 Drawing Command Stream

To efficiently draw a client scene, it's critical to create batches of drawing commands that share as much state as possible. In service of this, we represent the drawing stream as a first-class object in the engine, where elements in that stream are completely decoupled from the client scene's semantics.

Furthermore, to avoid exposing this low level "assembly language" to the render engine user, we propose a system of automatically converting render prims into this stream; caching generated items; and sorting them for highly efficient drawing.

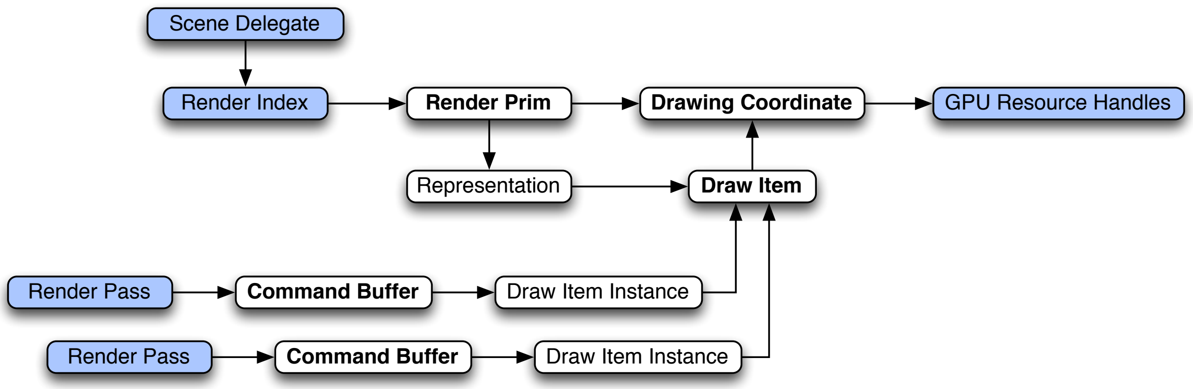


Figure 3: Drawing Command Buffers

4 Management, Tracking and Sharing of GPU Resources

The rendering task has complexity on the order of the number of low-level resources needed to draw the scene. Items managed by the registry are things like loaded texture images, allocated VBOs used for primvar data and drawing topology, compiled and linked GLSL shading programs, etc.

Resources are resolved by association with a descriptor. In the simplest case this descriptor look-up is a kind of hashing that allows the system to exploit possible resource sharing.

Separating the resources from the RenderIndex allows great opportunities for efficiency: e.g. texture images referenced at many points in the scene can resolve to a single loaded texture image, meshes sharing the same topology can share the same computed (or possibly serialized) topology tables, a single mesh might be displayed as both a smooth refined surface and an unrefined control hull, etc.

5 Drawing Coordinate and GLSL CodeGen

The "Drawing Coordinate" in Hydra represents a location of data on both the CPU and GPU. On the CPU side, we use it to generate GLSL shaders which stripe data through the shading pipeline. On the GPU side, we use it as an abstraction so that the client shader can fetch the data without knowing where the actual value is coming from. For example, HdGetColor() function could return constant color, face color or vertex color (Table 1).

Figure 4 shows the relationship between the DrawingCoordinate and aggregated buffers. Each entry of DrawingCoordinate points to the relative offsets within the aggregated buffers. Hydra exploits this level of indirection in many ways: data de-duplication (topology instancing, etc) can be expressed by sharing the same offsets, and per-instance frustum culling can be implemented by shuffling the instance index buffer.

The other benefit of having DrawingCoordinate is that it defines tessellation well, by declaring the shader inputs in this form.

Table 1: Drawing Coordinate

Coordinate	**Interpolation**	**Typical usages**
constant primvars	per object	transform visibility handedness
instance primvars	per instance	transform
instance index base	per instance	instance indirection
element primvars	per face/curve	face color material index
primitive buffer	per triangle/quad/patch	triangle to face mapping patch parameter ptex index
vertex primvars	per vertex	positions normals colors
shader parameters	per shader	kd, ks, transparency

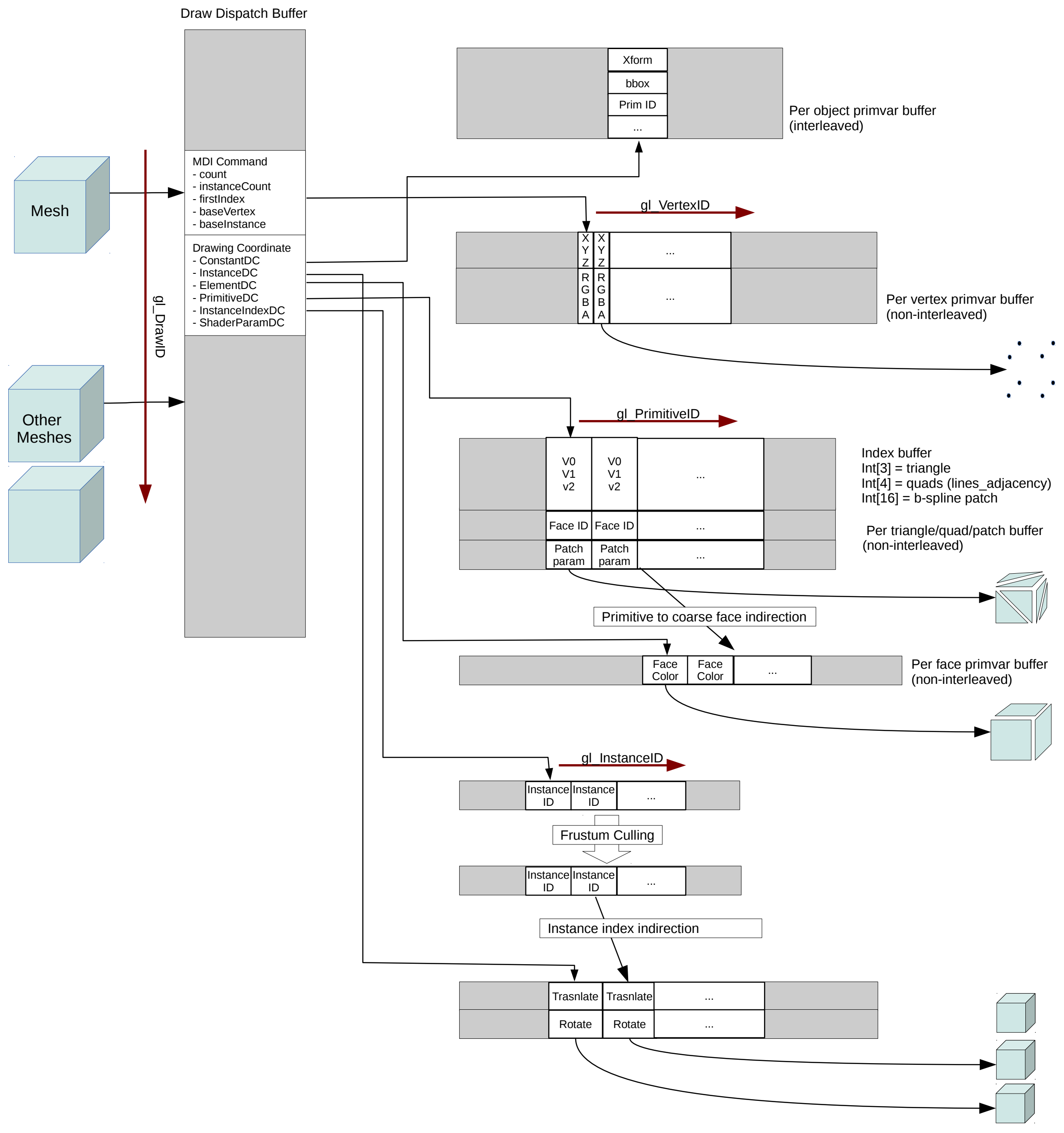


Figure 4: Buffer Aggregation and Drawing Coordinate

6 Multi Draw Indirect, Bindless Buffers

Hydra supports both immediate and indirect draws. In either case Hydra assorts drawing items into batches to minimize chaging GL states and buffer binding and unbidning overheads. In indirect draw mode, we construct a dispatch buffer which contains GPU draw commands and drawing coordinate for each command interleaved.

Primvars other than vertex primvar are fetched through shader storage buffer or bindless unform buffer if it's available on the runtime hardware.

7 Shading Interface

Hydra's shading system is designed to decouple the appearance shader code from the input signals. The client is required to provide just "surface shader" and "displacement shader" functions, and each function can access an RenderPrim's input signals transparently without knowing how they are defined and interpolated, including by instancing and by OpenSubdiv's tessellations.

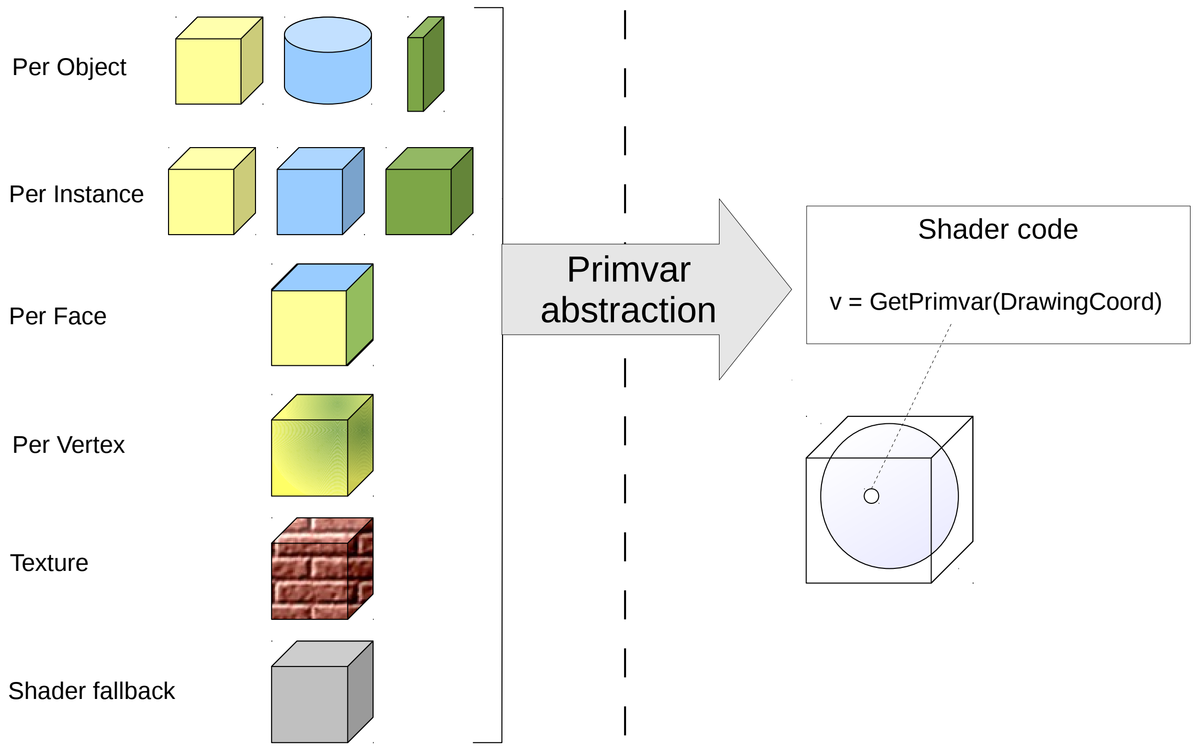


Figure 5: Primvar Abstraction

Figure 5 shows various cases of how primvars are defined on objects. Hydra's buffer aggregation creates the buckets which correspond to each primvar definition, and GLSL code generator composes the accessors to those primvars. The client shader code can use the same abstract interface to get those signals and can be agnostic about how they are defined on the objects.

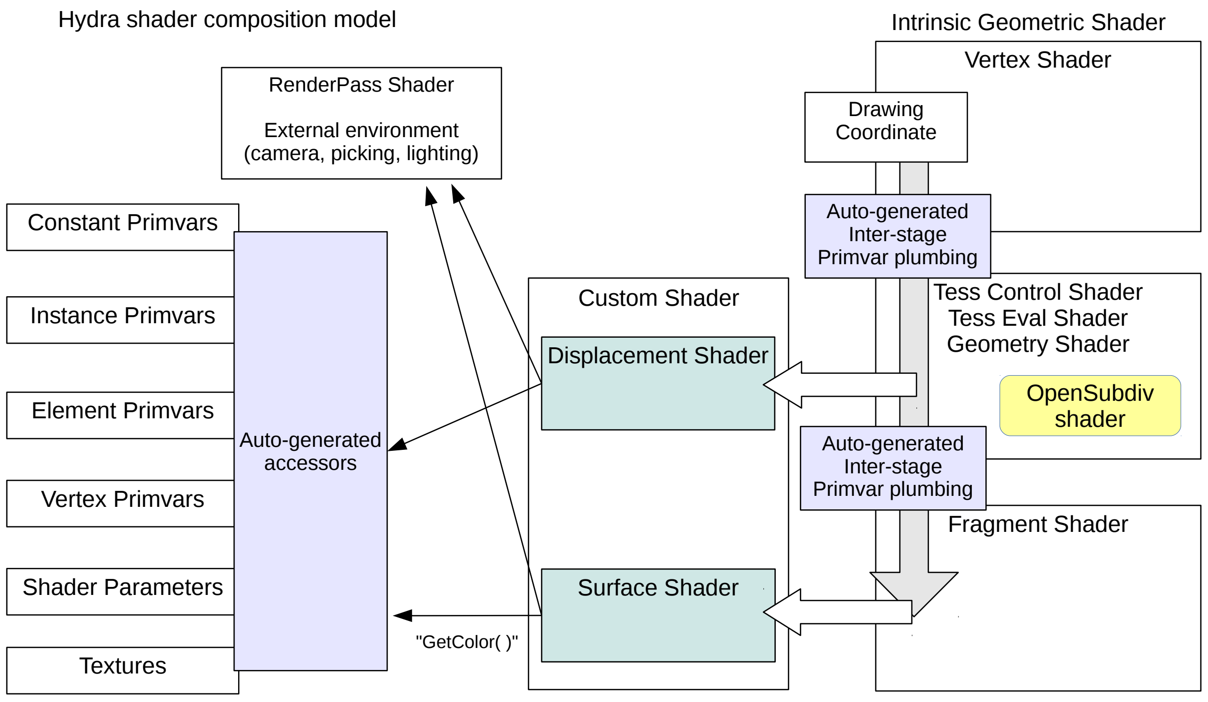


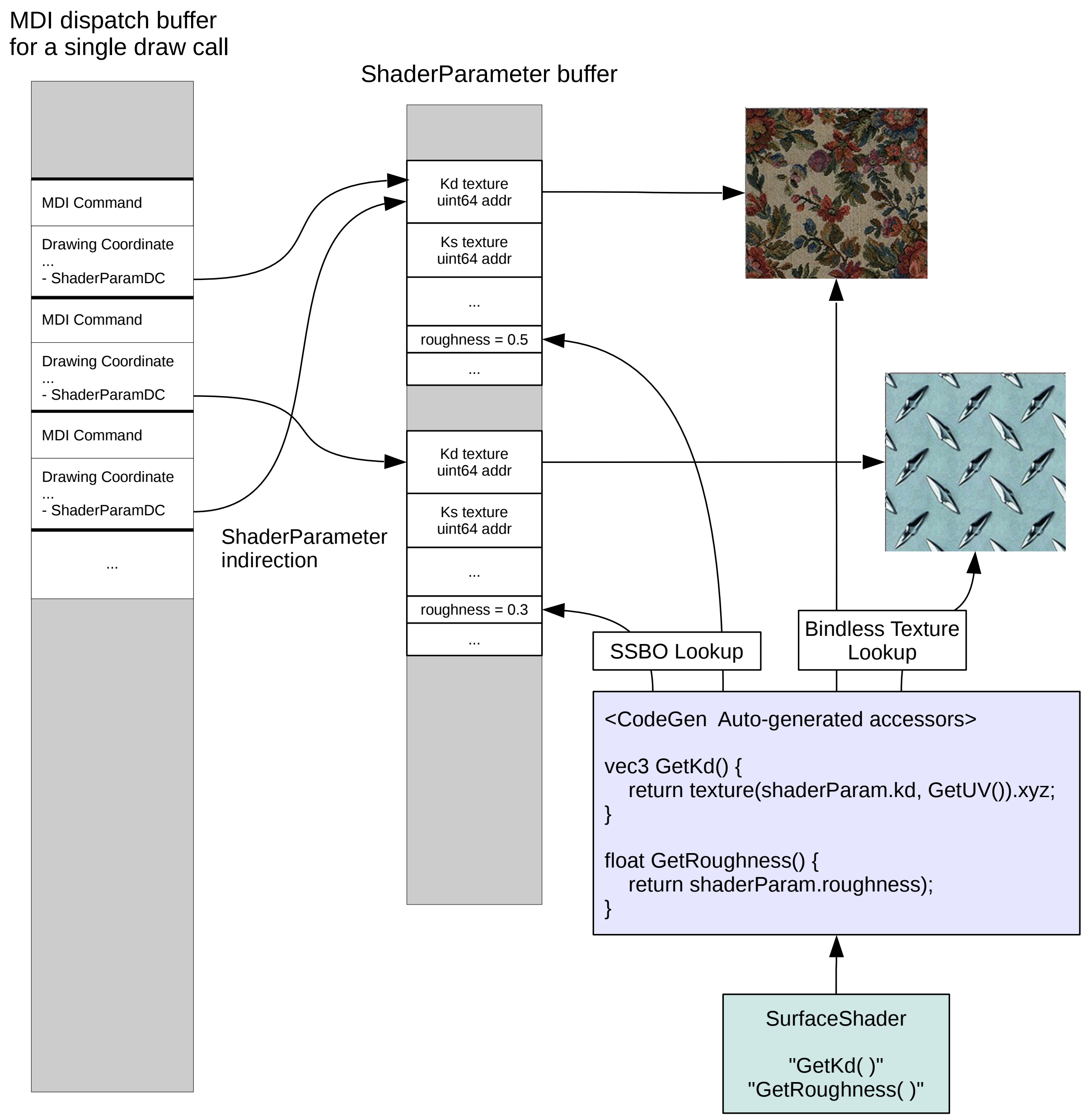
Figure 6: Shader CodeGen

Figure 6 describes the GLSL code generator in more detail. Hydra configures "Intrinsic Geometric Shader" for each RenderPrims. This shader is used as a template skeleton, which owns the main entries of each shader stage and defines GL primitives (triangles, quads, lines) as well as OpenSubdiv tessellated patches. It also handles the display stylings, such as wireframe, backface culling, flat and smooth shadings, etc.

To complete the pipeline configuration, the GLSL code generator scans the underlying buffer array of the RenderPrim, and it generates corresponding accessors for each buffer resources. For example, if a primvar is defined as a vertex primvar, the code generator generates code snippets of a vertex array declaration for that primvar along with the accessor method. If a primvar is defined as a constant or uniform primvar, it generates SSBO or bindless uniform declaration and its accessors. For the vertex primvars, the code generator also generates inter-stage plumbing code so that those primvars can be used in the following shader stages such as geometry and framgent shaders.

The external environments, like camera coordinate and lighting parameters, are given as shader code snippet from the RenderPass shader. The surface shader takes a position, a normal vector and a drawing coordinate and computes the resulting value with those auto-generated accessors of the input signals and the external environment parameters. One of the nice things of this decoupling is that the surface shader can be agnostic about not only instancing and tessellation but also an RenderPrim's handed-ness and display stylings such as backface culling and wireframe drawing because they are handled by intrinsic geometric shader code.

Picking is done mostly in the same way, the code generator generates the ID rendering shader which is given by the Renderpass shader for picking.



```
<CodeGen  Auto-generated accessors>

vec3 GetKd() {
    return texture(shaderParam.kd, GetUV()).xyz;
}

float GetRoughness() {
    return shaderParam.roughness);
}
```

Figure 7: MultiDrawIndirect + Bindless texture

This mechanism can be extended to incorporate OpenGL's Multi-Draw-Indirect and bindless texture API together for further optimized rendering (Figure 7).

Shader parameters for each RenderPrim can be fetched from the shader parameter buffer using shader DC. When a shader input signal is defined by a texture, the bindless GPU address of the texture is stored in the shader parameter buffer. The auto-generated shader code provides an abstract interface to fetch the texture through those addresses. As a consequence, hydra can aggregate multiple objects into a single drawcall together, which uses more than the number of physical texture unit in total. Note that the drawing batches are configured so that all the participating RenderPrims have the same parameter topology (i.e. how the primvars are defined and interpolated). For the example shown in Figure 7, Kd and Ks are defined as textures, whereas the roughness is given as a constant primvar. The RenderPrims which have same primvar definition can be batched together and drawn in a single drawcall using the same auto-generated shader program, MDI and bindless.

Hydra leverages the combination of these techniques to draw massive amount of objects with extremely low number of drawcalls.

8 Conclusion

The Hydra render engine was developed explicitly to support feature film workflows and assets. Decoupling the engine from the scenegraph allows for high fidelity and high performance preview in diverse scenarios and tools, while formulating the internals in terms of the RenderMan data model allows for direct final-frame assets to be used in real-time contexts. At a low level, this performance is made possible by reducing driver overhead via modern graphics APIs available in desktop OpenGL, but the architecture has also been designed with an eye toward forthcoming explicit graphics APIs, such as Vulkan, DX12, and Metal. As the realms of offline and real-time rendering converge, we continue to push for real-time approximations of final-frame fidelity.

OpenSubdiv 3.0 introduction

by Takahito Tejima

Figure 1: InsideOut©Disney/Pixar 2015

Abstract

Since OpenSubdiv 1.0 was initially released as an open source project in 2012, it has been widely adapted into various DCC tools and applications and is becoming an industry standard. We have gotten a huge response and feedback from many collaborators, and DreamWorks Animations is one of the biggest contributors. With their generous cooperation, we released a major update to 3.0 this year. OpenSubdiv 3.0 represents a landmark release, with very profound changes to the core algorithms. While providing faster, more efficient, and more flexible subdivision code remains our principal goal, OpenSubdiv 3.0 introduces many improvements that constitute a fairly radical departure from our previous versions.

This release not only focuses internal data structure, but also simplifies a lot of client facing APIs that make integration easy and very efficient. We implemented a unified patch tessellation shader which allows us to draw entire subdivision surface geometry with a single GL draw-call, while we're still getting benefits from sparseness of the adaptively refined patch representation.

Note: The latest version of this document is available online at http://graphics.pixar.com/opensubdiv/

1 Subdivision Surfaces

1.1 Introduction

The most common way to model complex smooth surfaces is by using a patchwork of bicubic patches such as BSplines or NURBS.

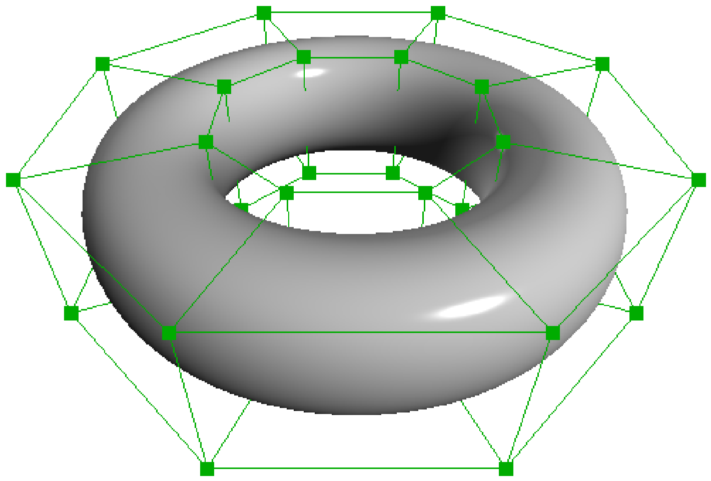


Figure 2: A torus modeled by B-Spline patches

However, while they do provide a reliable smooth limit surface definition, bi-cubic patch surfaces are limited to 2-dimensional topologies, which only describe a very small fraction of real-world shapes. This fundamental parametric limitation requires authoring tools to implement at least the following functionalities:

- smooth trimming
- seams stitching

Both trimming and stitching need to guarantee the smoothness of the model both spatially and temporally as the model is animated. Attempting to meet these requirements introduces a lot of expensive computations and complexity.

Subdivision surfaces on the other hand can represent arbitrary topologies, and therefore are not constrained by these difficulties.

1.2 Arbitrary Topology

A subdivision surface, like a parametric surface, is described by its control mesh of points. The surface itself can approximate or interpolate this control mesh while being piecewise smooth. But where polygonal surfaces require large numbers of data points to approximate being smooth, a subdivision surface is smooth - meaning that polygonal artifacts are never present, no matter how the surface animates or how closely it is viewed.

Ordinary cubic B-spline surfaces are rectangular grids of tensor-product patches. Subdivision surfaces generalize these to control grids with arbitrary connectivity.

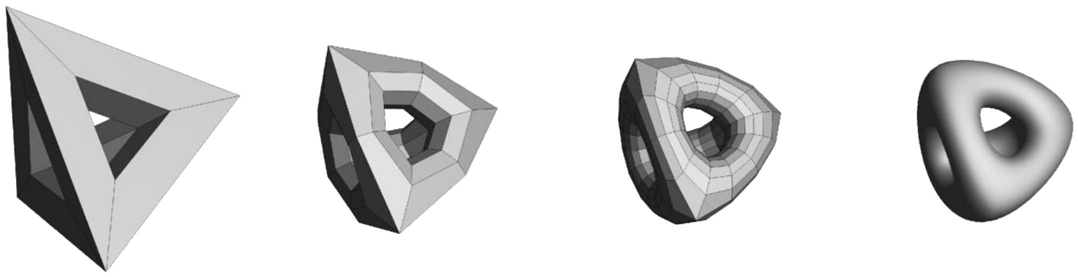


Figure 3: subdivisions of topologically complex mesh

1.3 Uniform Subdivision and Adaptive Subdivision

Applies a uniform refinement scheme to the coarse faces of a mesh. The mesh converges closer to the limit surface with each iteration of the algorithm.

Applies a progressive refinement strategy to isolate irregular features. The resulting vertices can be assembled into bi-cubic patches defining the limit surface.

Feature adaptive refinement can be much more economical in terms of time and memory use, but the best method to use depends on application needs.

The following table identifies several factors to consider:

Table 2: Uniform vs Adaptive

Uniform	Feature Adaptive
Exponential geometry growth	Geometry growth close to linear and occuring only in the neighborhood of isolated topological features
Current implementation only produces bi-linear patches for uniform refinement	Current implementation only produces bi-cubic patches for feature adaptive refinement
All face-varying interpolation rules supported at refined vertex locations	Currently, only bi-linear face-varying interpolation is supported for bi-cubic patches

1.4 Boundary Interpolation Rules

Boundary interpolation rules control how boundary edges and vertices are interpolated.

The following rule sets can be applied to vertex data interpolation:

Table 3: Boundary Interpolation Rules

Mode	Behavior
VTX_BOUNDARY_NONE	No boundary edge interpolation should occur; instead boundary faces are tagged as holes so that the boundary edge-chain continues to support the adjacent interior faces but is not considered to be part of the refined surface
VTX_BOUNDARY_EDGE_ONLY	All the boundary edge-chains are sharp creases; boundary vertices are not affected
VTX_BOUNDARY_EDGE_AND_CORNER	All the boundary edge-chains are sharp creases and boundary vertices with exactly one incident face are sharp corners

On a grid example:

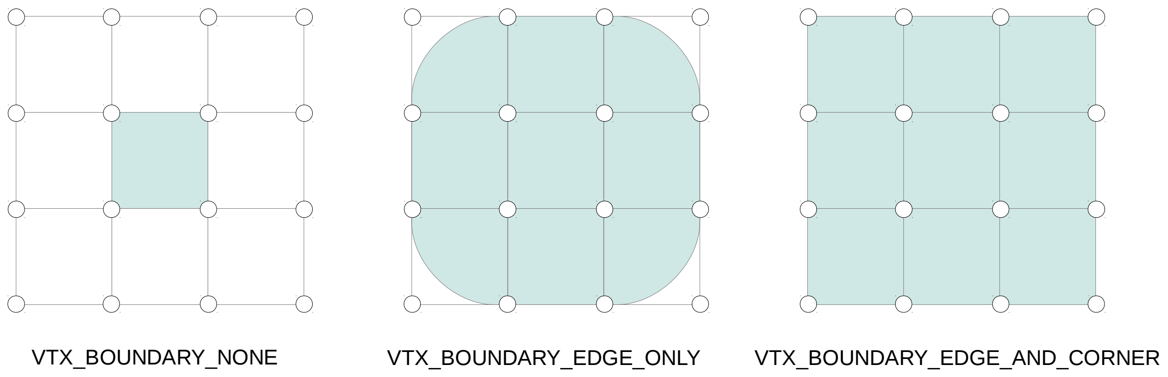


Figure 4: Boundary interpolation rules

1.5 Face-Varying Interpolation Rules

Face-varying data is used when discontinuities are required in the data over the surface – mostly commonly the seams between disjoint UV regions. Face-varying data can follow the same interpolation behavior as vertex data, or it can be constrained to interpolate linearly around selective features from corners, boundaries, or the entire interior of the mesh.

The following rules can be applied to face-varying data interpolation – the ordering here applying progressively more linear constraints:

Table 4: Face-Varying interpolation rules

Mode	Behavior
FVAR_LINEAR_NONE	smooth everywhere the mesh is smooth
FVAR_LINEAR_CORNERS_ONLY	sharpen (linearly interpolate) corners only
FVAR_LINEAR_CORNERS_PLUS1	CORNERS_ONLY + sharpening of junctions of 3 or more regions
FVAR_LINEAR_CORNERS_PLUS2	CORNERS_PLUS1 + sharpening of darts and concave corners
FVAR_LINEAR_BOUNDARIES	linear interpolation along all boundary edges and corners
FVAR_LINEAR_ALL	linear interpolation everywhere (boundaries and interior)

These rules cannot make the interpolation of the face-varying data smoother than that of the vertices. The presence of sharp features of the mesh created by sharpness values, boundary interpolation rules, or the subdivision scheme itself (e.g. Bilinear) take precedence.

All face-varying interpolation modes illustrated in UV space using the catmark_fvar_bound1 regression shape – a simple 4x4 grid of quads segmented into three UV regions (their control point locations implied by interpolation in the FVAR_LINEAR_ALL case):

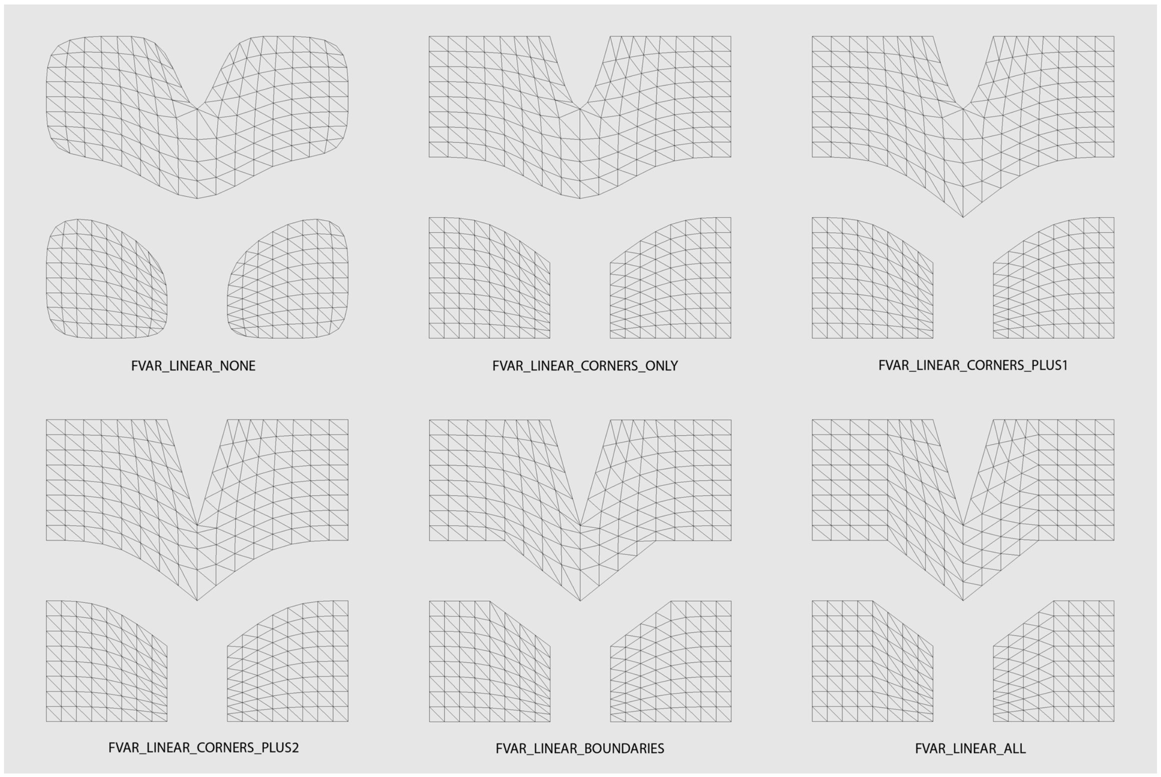


Figure 5: facevarying rules example of catmark_fvar_bound1

1.6 Semi-Sharp Creases

It is possible to modify the subdivision rules to create piecewise smooth surfaces containing infinitely sharp features such as creases and corners. As a special case, surfaces can be made to interpolate their boundaries by tagging their boundary edges as sharp.

However, we've recognized that real world surfaces never really have infinitely sharp edges, especially when viewed sufficiently close. To this end, we've added the notion of semi-sharp creases, i.e. rounded creases of controllable sharpness. These allow you to create features that are more akin to fillets and blends. As you tag edges and edge chains as creases, you also supply a sharpness value that ranges from 0-10, with sharpness values ¿=10 treated as infinitely sharp.

It should be noted that infinitely sharp creases are really tangent discontinuities in the surface, implying that the geometric normals are also discontinuous there. Therefore, displacing along the normal will likely tear apart the surface along the crease. If you really want to displace a surface at a crease, it may be better to make the crease semi-sharp.

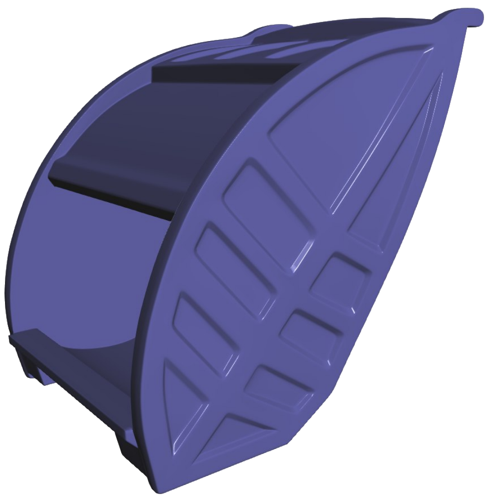


Figure 6: A shape modeled with semi-sharp creases

1.7 Chaikin Rule

Chaikin's curve subdivision algorithm improves the appearance of multi-edge semi-sharp creases with varying weights. The Chaikin rule interpolates the sharpness of incident edges.

Table 5: Chaikin rule

Mode	Behavior
CREASE_UNIFORM	Apply regular semi-sharp crease rules
CREASE_CHAIKIN	Apply "Chaikin" semi-sharp crease rules

Example of contiguous semi-sharp creases interpolation:

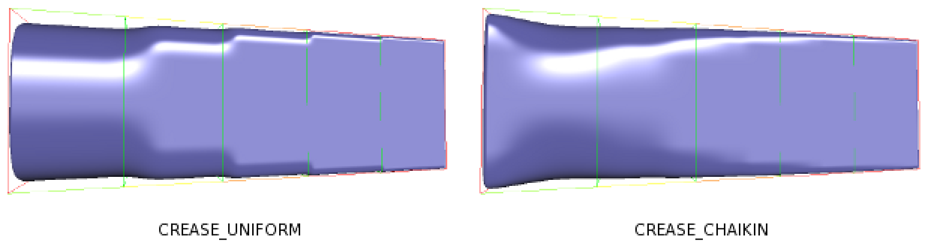


Figure 7: Chaikin rule example

1.8 "Triangle Subdivision" Rule

The triangle subdivision rule is a rule added to the Catmull-Clark scheme that can be applied to all triangular faces; this rule was empirically determined to make triangles subdivide more smoothly. However, this rule breaks the nice property that two separate meshes can be joined seamlessly by overlapping their boundaries; i.e. when there are triangles at either boundary, it is impossible to join the meshes seamlessly

Table 6: "Triangle Subdivision" Rule

Mode	Behavior
TRI_SUB_CATMARK	Default Catmark scheme weights
TRI_SUB_SMOOTH	"Smooth triangle" weights

Cylinder example :

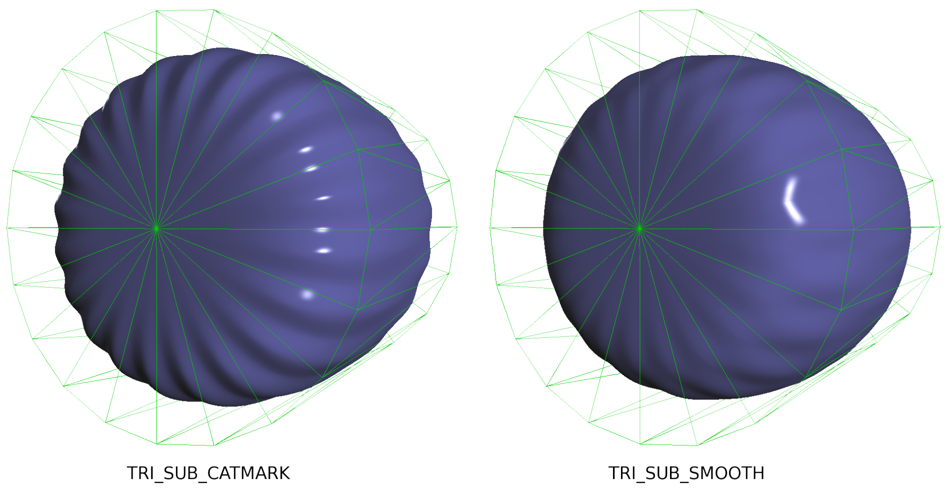


Figure 8: Triangle subdivision rule

1.9 Manifold vs Non-Manifold Geometry

Continuous limit surfaces generally require that the topology be a two-dimensional manifold for the limit surface to be unambiguous. It is possible (and sometimes useful, if only temporarily) to model non-manifold geometry and so create surfaces whose limit is not as well-defined.

The following examples show typical cases of non-manifold topological configurations.

1.9.1 Non-Manifold Fan

This "fan" configuration shows an edge shared by 3 distinct faces.

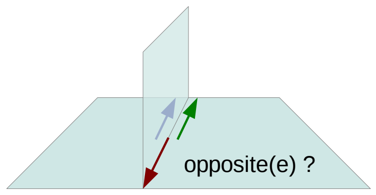


Figure 9: Non-Manifold geometry

With this configuration, it is unclear which face should contribute to the limit surface (assuming it is singular) as three of them share the same edge. Fan configurations are not limited to three incident faces: any configuration where an edge is shared by more than two faces incurs the same problem.

These and other regions involving non-manifold edges are dealt with by considering regions that are "locally manifold". Rather than a single limit surface through this problematic edge with its many incident faces, the edge locally partitions a single limit surface into more than one. So each of the three faces here will have their own (locally manifold) limit surface – all of which meet at the shared edge.

1.9.2 Non-Manifold Disconnected Vertex

A vertex is disconnected from any edge and face.

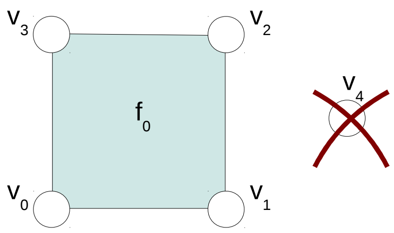


Figure 10: Non-Manifold disconnected vertex

This case is fairly trivial: there is a very clear limit surface for the four vertices and the face they define, but no possible way to exact a limit surface from the disconnected vertex.

While the vertex does not contribute to any limit surface, it may not be completely irrelevant though. Such vertices may be worth retaining during subdivision (if for no other reason than to preserve certain vertex ordering) and simply ignored when it comes time to consider the limit surface.

2 API Overview

API Layers

OpenSubdiv is structured as a set of layered libraries. This structure facilitates operation on a variety of computing resources, and allows developers to only opt-in to the layers and feature sets that they require. From a top-down point of view, OpenSubdiv is comprised of several layers, some public, and some private.

Layers list:

Table 7: API Layer list

Sdc (Subdivision Core)	The lowest level layer, implements the core subdivision details to facilitate the generation of consistent results. Most cases will only require the use of simple public types and constants from Sdc.
Vtr (Vectorized Topological Representation)	A suite of classes to provide an intermediate representation of topology that supports efficient refinement. Vtr is intended for internal use only.
Far (Feature Adaptive Representation)	The central interface that processes client-supplied geometry and turns it into a serialized data representation ready for parallel processing in Osd. Far also provides a fully-featured single-threaded implementation of subdivision interpolation algorithms.
Osd (OpenSubdiv cross platform)	A suite of classes to provide parallel subdivision kernels and drawing utilities on a variety of platforms such as TBB, CUDA, OpenCL, GLSL and DirectX.

Client mesh data enters the API through the Far layer. Typically, results will be collected from the Osd layer. However, it is possible to use functionality from Far without introducing any dependency on Osd.

Although there are several entry-points to provide topology and primitive variable data to OpenSubdiv, eventually everything must pass through the private Vtr and Sdc representations for topological analysis.

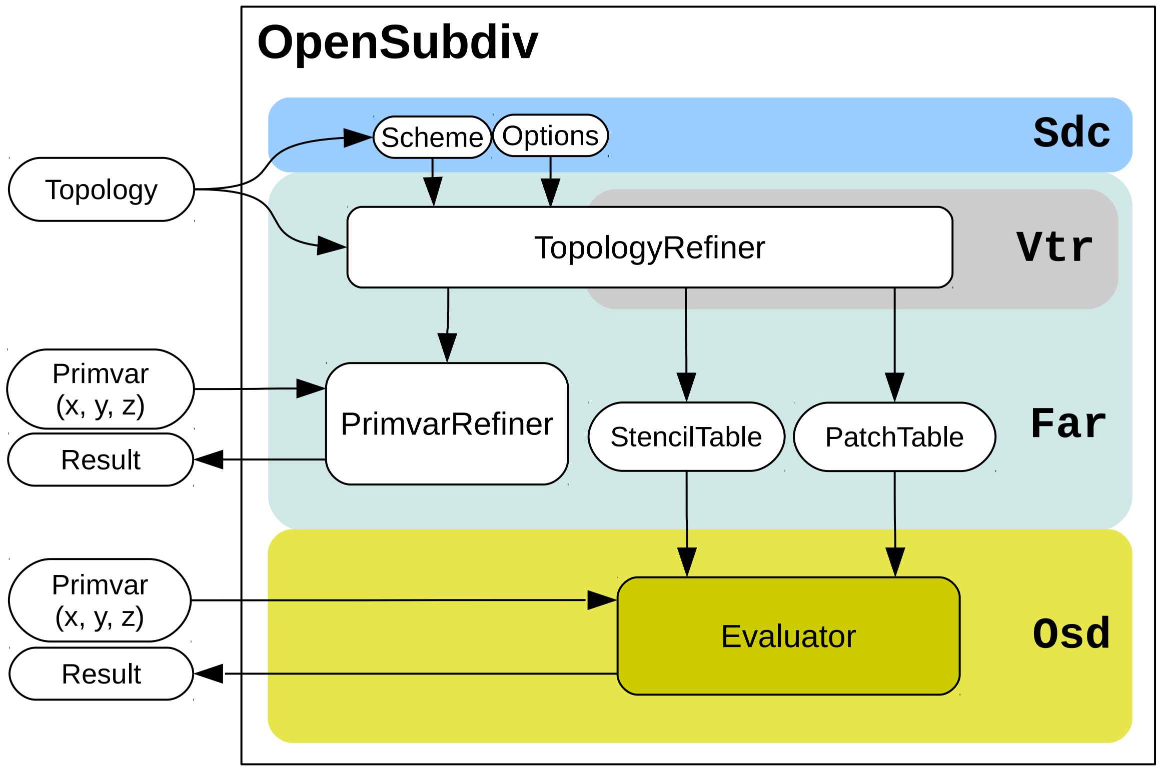


Figure 11: API layers

2.1 Using the Right Tools

OpenSubdiv's tiered interface offers a lot flexibility to make your application both fast and robust. Because navigating through the large collection of classes and features can be challenging, here are use cases that should help sketch the broad lines of going about using subdivisions in your application.

General client application requirements:

Table 8: API Layer list

Surface Limit	For some applications, a polygonal approximation of the smooth surface is enough. Others require C 2 continuous differentiable bi-cubic patches (ex: deformable displacement mapping, smooth normals and semi-sharp creases...)
Deforming Surface	Applications such as off-line image renderers often process a single frame at a time. Others, such as interactive games need to evaluate deforming character surface every frame. Because we can amortize many computations if the topology of the mesh does not change, OpenSubdiv provides 'stencil tables' in order to leverage subdivision refinement into a pre-computation step.
Multi-threading	OpenSubdiv also provides dedicated interfaces to leverage parallelism on a wide variety of platforms and API standards, including both CPUs and GPUs.
GPU Draw	If the application requires interactive drawing on screen, OpenSubdiv provides several back-end implementations, including D3D11 and OpenGL. These back-ends provide full support for programmable shading.

2.2 Use case 1: Simple refinement

The following example shows the most simple case to get your mesh refined uniformly.

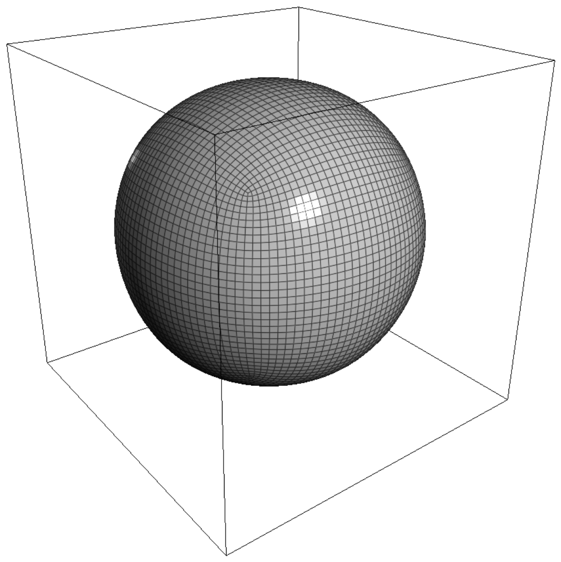


Figure 12: Use case 1: Simple refinement

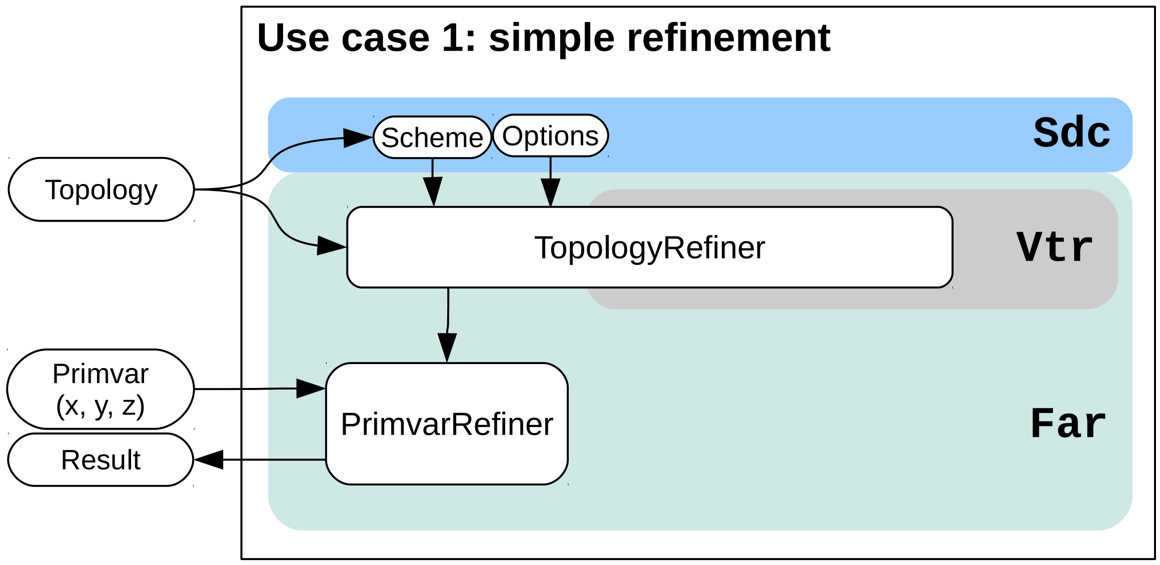


Figure 13: Use case 1: Simple refinement

1. Define a class for the primvar you want to refine. It's required to have Clear() and AddWithWeight() functions.

```
struct Vertex {
    void Clear() { x = y = z = 0; }
    void AddWithWeight(Vertex const &src, float weight) {
        x += weight * src.x;
        y += weight * src.y;
        z += weight * src.z;
    }
    float x, y, z;
};
```

2. Instantiate a `Far::TopologyRefiner` from the `Far::TopologyDescriptor`.

```
Far::TopologyDescriptor desc;
desc.numVertices         = <the number of vertices>
desc.numFaces            = <the number of faces>
desc.numVertsPerFace     = <array of the number of verts per face>
desc.vertIndicesPerFace  = <array of vert indices>

Far::TopologyRefiner * refiner = Far::TopologyRefinerFactory<Descriptor>::Create(desc);
```

3. Call RefineUniform() to refine the topology up to 'maxlevel'.

```
refiner->RefineUniform(Far::TopologyRefiner::UniformOptions(maxlevel));
```

4. Interpolate vertex primvar data at 'level' using Far::PrimvarRefiner

```
Far::PrimvarRefiner primvarRefiner(*refiner);
```

```
Vertex const *src = <coarse vertices>
Vertex *dst       = <refined vertices>

primvarRefiner.Interpolate(level, src, dst);
```

5. The topology at the refined level can be obtained from Far::TopologyLevel.

```
Far::TopologyLevel const & refLastLevel = refiner->GetLevel(maxlevel);

int nverts = refLastLevel.GetNumVertices();
int nfaces = refLastLevel.GetNumFaces();

for (int face = 0; face < nfaces; ++face) {
    Far::ConstIndexArray fverts = refLastLevel.GetFaceVertices(face);

    // do something with dst and fverts
}
```

2.3 Use case 2: GL adaptive tessellation drawing of animating mesh

The next example is showing how to draw adaptive tessellated patches in GL using OpenSubdiv. The osd layer helps you to interact with GL and other device specific APIs. Also for an efficient refinement of animating mesh on a static topology, we create a stencil table to refine the positions changing over time.

The following example code uses an Osd::GLMesh utility class which composites a stencil table, patch table, vertex buffer and evaluator in osd layer. You can also use those classes independently.

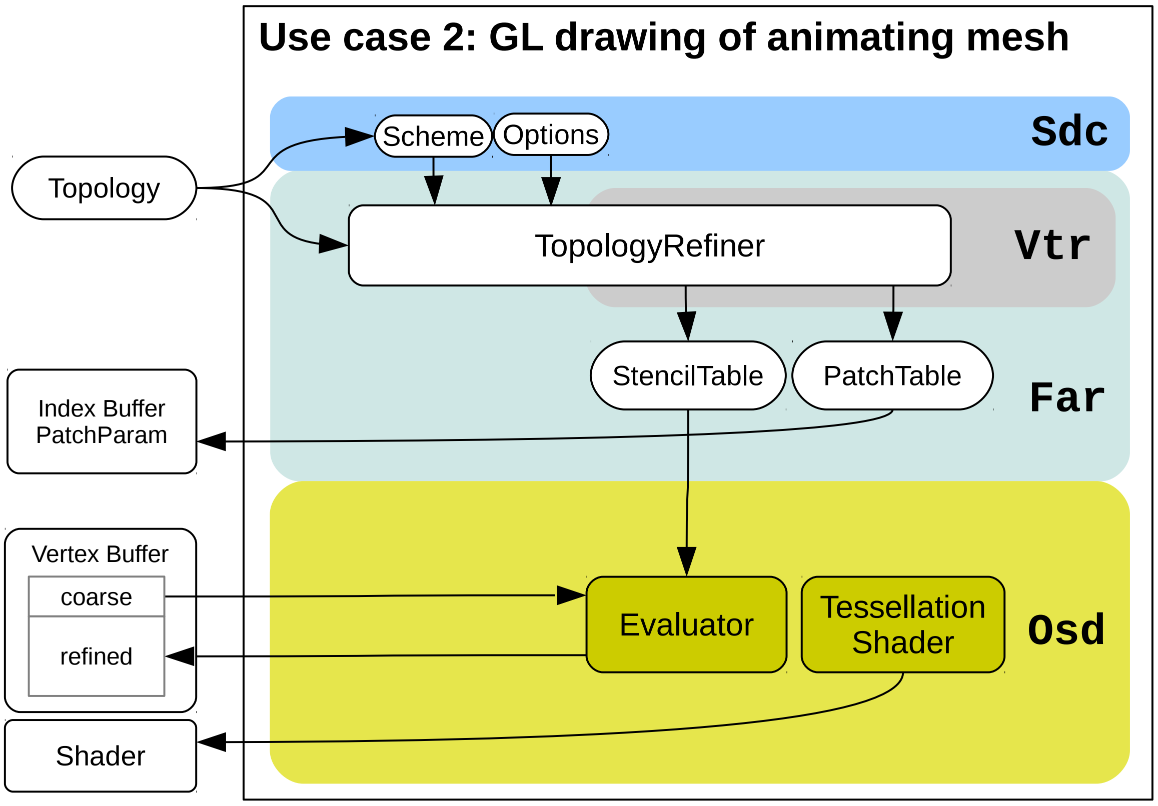


Figure 14: Use case 2: GL adaptive tessellation drawing

1. Instantiate a `Far::TopologyRefiner` from the `Far::TopologyDescriptor`, same as usecase 1.
2. Setup Osd::Mesh. In this example we use b-spline endcap.

```
int numVertexElements = 3; // x, y, z

Osd::MeshBitset bits;
bits.set(Osd::MeshAdaptive, true);           // set adaptive
bits.set(Osd::MeshEndCapBSplineBasis, true); // use b-spline basis patch for endcap.

Osd::GLMeshInterface *mesh = new Osd::Mesh<Osd::CpuGLVertexBuffer, Far::StencilTable,
                                           Osd::CpuEvaluator, Osd::GLPatchTable>
                                    (refiner, numVertexElements, 0, level, bits);
```

3. Update coarse vertices and refine (Osd::Mesh::Refine() calls Osd::CpuEvaluator::EvalStencils())

```
mesh->UpdateVertexBuffer(&vertex[0], 0, nverts);
mesh->Refine();
```

4. Bind index buffer, PatchParamBuffer and vertex buffer.

```
// index buffer
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, mesh->GetPatchTable()->GetPatchIndexBuffer());

// vertex buffer
glBindBuffer(GL_ARRAY_BUFFER, mesh->BindVertexBuffer());
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, numVertexElements, GL_FLOAT, GL_FALSE,
                      numVertexElements*sizeof(float), 0);

// patch param buffer
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_BUFFER, mesh->GetPatchTable()->GetPatchParamTextureBuffer());
```

5. Draw. Since we use b-spline endcaps in this example, there is only one PatchArray in the patch table. You may need to iterate patch arrays as you use other type of endcap. To configure GLSL program for each patch type, see osd shader interface for more details.

```
Osd::PatchArray const & patch = mesh->GetPatchTable()->GetPatchArrays()[0];
Far::PatchDescriptor desc = patch.GetDescriptor();

int numVertsPerPatch = desc.GetNumControlVertices();  // 16 for B-spline patches
glUseProgram(BSplinePatchProgram);
glPatchParameteri(GL_PATCH_VERTICES, numVertsPerPatch);
glDrawElements(GL_PATCHES, patch.GetNumPatches() * numVertsPerPatch,
               GL_UNSIGNED_INT, 0);
```

6. As the mesh animates, repeat from step 3 to update positions, refine, and draw. See glViewer and other examples for more complete usage.

3 Sdc Overview

3.1 Subdivision Core (Sdc)

Sdc is the lowest level layer in OpenSubdiv. Its intent is to separate the core subdivision details from any particular representation of a mesh (it was previously bound to Hbr) to facilitate the generation of consistent results with other mesh representations, both internal and external to OpenSubdiv.

The functionality can be divided roughly into three sections:

- types, traits and options for the supported subdivision schemes
- computations required to support semi-sharp creasing
- computations for mask weights of subdivided vertices for all schemes

For most common usage, familiarity with only the first of these is necessary – primarily the use of public types and constants for the choice of subdivision scheme and its associated options. The latter two provide the basis for a more comprehensive implementation of subdivision, which requires considerably more understanding and effort.

Overall, the approach was to extract the functionality at the lowest level possible. In some cases, the implementation is not far from being simple global functions. The intent was to start at a low level and build any higher level functionality as needed. What exists now is functional for ongoing development and anticipated needs within OpenSubdiv for the near future.

The intent of Sdc is to provide the building blocks for OpenSubdiv and its clients to efficiently process the specific set of supported subdivision schemes. It is not intended to be a general framework for defining customized subdivision schemes.

3.2 Types, Traits and Options

The most basic type is the enum Sdc::SchemeType that identifies the fixed set of subdivision schemes supported by OpenSubdiv: Bilinear, Catmark and Loop. With this alone, we intend to avoid all dynamic casting issues related to the scheme by simply adding members to the associated subclasses for inspection.

In addition to the type enum itself, a class defining a fixed set of traits associated with each scheme is provided. While these traits are available as static methods in the interface of a class supporting more functionality for each scheme (to be described shortly), the SchemeTypeTraits provide queries of the traits for a variable of type Sdc::SchemeType – enabling parameterization of code by the value of a trait without templates or virtual inheritance (a simple internal table of traits is constructed and trivially indexed).

The second contribution is the collection of all variations in one place that can be applied to the subdivision schemes, i.e. the boundary interpolation rules, creasing method, edge subdivision choices, etc. The fact that these are all declared in one place alone should help clients see the full set of variations that are possible.

A simple Options struct (a set of bitfields) aggregates all of these variations into a single object (the equivalent of an integer in this case) that are passed around to other Sdc classes and/or methods and are expected to be used at a higher level both within OpenSubdiv and externally. By aggregating the options and passing them around as a group, it allows us to extend the set easily in future without the need to rewire a lot of interfaces to accommodate the new choice. Clients can enable new choices at the highest level and be assured that they will propagate to the lowest level where they are relevant.

Unlike other "options" structs used elsewhere to specify variations of a particular method, Sdc::Options defines all options that affect the shape of the underlying limit surface of a subdivision mesh. Other operations at higher levels in the library may have options that approximate the shape and so create a slightly different appearance, but Sdc::Options is a fundamental part of the definition of the true limit surface.

3.3 Creasing support

Since the computations involved in the support of semi-sharp creasing are independent of the subdivision scheme, the goal in Sdc was to encapsulate all related creasing functionality in a similarly independent manner. Computations involving sharpness values are also much less dependent on topology – there are vertices and edges with sharpness values, but knowledge of faces or boundary edges is not required, – so the complexity of topological neighborhoods required for more scheme-specific functionality is arguably not necessary here.

Creasing computations have been provided as methods defined on a Crease class that is constructed with a set of Options. Its methods typically take sharpness values as inputs and compute a corresponding set of sharpness values as a result. For the "Uniform" creasing method (previously known as "Normal"), the computations may be so trivial as to question whether such an interface is worth it, but for "Chaikin" or other schemes in the future that are non-trivial, the benefits should be clear. Functionality is divided between both uniform and non-uniform, so clients have some control over avoiding unnecessary overhead, e.g. non-uniform computations typically require neighboring sharpness values around a vertex, while uniform does not.

Also included as part of the Crease class is the Rule enum – this indicates if a vertex is Smooth, Crease, Dart or Corner (referred to as the "mask" in Hbr) and is a function of the sharpness values at and around a vertex. Knowing the Rule for a vertex can accelerate mask queries, and the Rule can often be inferred based on the origin of a vertex (e.g. it originated from the middle of a face, was the child of a Smooth vertex, etc.).

Methods are defined for the Crease class to:

- subdivide edge and vertex sharpness values
- determine the Rule for a vertex based on incident sharpness values
- determine the transitional weight between two sets of sharpness values

Being all low-level and working directly on sharpness values, it is a client's responsibility to coordinate the application of any hierarchical crease edits with their computations.

Similarly, in keeping with this as a low-level interface, values are passed as primitive arrays. This follows the trend in OpenSubdiv of dealing with data of various kinds (e.g. weights, component indices, now sharpness values, etc.) in small contiguous sets of values. In most internal cases we can refer to a set of values or gather what will typically be a small number of values on the stack for temporary use.

3.4 Scheme-specific support

While the SchemeTypeTraits class provides traits for each subdivision scheme supported by OpenSubdiv (i.e. Bilinear, Catmark and Loop), the Scheme class provides these more directly, Additionally, the Scheme class provides methods for computing the various sets of weights used to compute new vertices resulting from subdivision. The collection of weights used to compute a single vertex at a new subdivision level is typically referred to as a "mask". The primary purpose of the Scheme class is to provide such masks in a manner both general and efficient.

Each subdivision scheme has its own values for its masks, and each are provided as specializations of the template class `Scheme<SchemeType TYPE>`. The intent is to minimize the amount of code specific to each scheme.

The computation of mask weights for subdivided vertices is the most significant contribution of Sdc. The use of semi-sharp creasing with each non-linear subdivision scheme complicates what are otherwise simple masks determined solely by the topology, and packaging that functionality to achieve both the generality and efficiency desired has been a challenge.

Mask queries are defined in the Scheme class template, which has specializations for each of the supported subdivision schemes. Mask queries are defined in terms of interfaces for two template parameters: the first defining the topological neighborhood of a vertex, and a second defining a container in which to gather the individual weights:

```
template <typename FACE, typename MASK>
void ComputeFaceVertexMask(FACE const& faceNeighborhood, MASK& faceVertexMask, ...) const;
```

Each mask query is expected to call methods defined for the FACE, EDGE or VERTEX classes to obtain the information they require ; typically these methods are simple queries about the topology and associated sharpness values. Clients are free to use their own mesh representations to gather the requested information as quickly as possible, or to cache some subset as member variables for immediate inline retrieval.

In general, the set of weights for a subdivided vertex is dependent on the following:

- the topology around the parent component from which the vertex originates
- the type of subdivision Rule applicable to the parent component
- the type of subdivision Rule applicable to the new child vertex
- a transitional weight blending the effect between differing parent and child rules

  This seems fairly straight-forward, until we look at some of the dependencies involved:

- the parent Rule requires the sharpness values at and around the parent component
- the child Rule requires the subdivided sharpness values at and around the new child vertex (though it can sometimes be trivially inferred from the parent)
- the transitional weight between differing rules requires all parent and child sharpness values

Clearly the sharpness values are inspected multiple times and so it pays to have them available for retrieval. Computing them on an as-needed basis may be simple for uniform creasing, but a non-uniform creasing method requires traversing topological neighborhoods, and that in addition to the computation itself can be costly.

The point here is that it is potentially unreasonable to expect to evaluate the mask weights completely independent of any other consideration. Expecting and encouraging the client to have subdivided sharpness values first, for use in more than one place, is therefore recommended.

The complexity of the general case above is also unnecessary for most vertices. Any client using Sdc typically has more information about the nature of the vertex being subdivided and much of this can be avoided – particularly for the smooth interior case that often dominates. More on that in the details of the Scheme classes.

Given that most of the complexity has been moved into the template parameters for the mask queries, the Scheme class remains fairly simple. Like the Crease class, it is instantiated with a set of Options to avoid them cluttering the interface. It is currently little more than a few methods for the limit and refinement masks for each vertex type, plus the few fixed traits of the scheme as static methods.

The mask queries have been written in a way that greatly simplifies the specializations required for each scheme. The generic implementation for both the edge-vertex and vertex-vertex masks take care of all of the creasing logic, requiring only a small set of specific masks to be assigned for each Scheme: smooth and crease masks for an edge-vertex, and smooth, crease and corner masks for a vertex-vertex. Other than the Bilinear case, which will specialize the mask queries to trivialize them for linear interpolation, the specializations for each Scheme should only require defining this set of masks – and with two of them common (edge-vertex crease and vertex-vertex corner) the Catmark scheme only needs to define three.

3.4.1 The FACE, EDGE and VERTEX interfaces

Mask queries require an interface to a topological neighborhood, currently labeled FACE, EDGE and VERTEX. This naming potentially implies more generality than intended, as such classes are only expected to provide the methods required of the mask queries to compute its associated weights. While all methods must be defined, some may rarely be invoked, and the client has considerable flexibility in the implementation of these: they can defer some evaluations lazily until required, or be pro-active and cache information in member variables for immediate access.

An approach discussed in the past has alluded to iterator classes that clients would write to traverse their meshes. The mask queries would then be parameterized in terms of a more general and generic mesh component that would make use of more general traversal iterators. The advantage here is the iterators are written once, then traversal is left to the query and only what is necessary is gathered. The disadvantages are that clients are forced to write these to do anything, getting them correct and efficient may not be trivial (or possible in some cases), and that the same data (e.g. subdivided sharpness) may be gathered or computed multiple times for different purposes.

The other extreme was to gather everything possible required at once, but that is objectionable. The approach taken here provides a reasonable compromise between the two. The mask queries ask for exactly what they want, and the provided classes are expected to deliver it as efficiently as possible. In some cases the client may already be storing it in a more accessible form and general topological iteration can be avoided.

The information requested of these classes in the three mask queries is as follows:

For FACE

- the number of incident vertices

For EDGE

- the number of incident faces
- the sharpness value of the parent edge
- the sharpness values of the two child edges
- the number of vertices per incident face

For VERTEX

- the number of incident faces
- the number of incident edges
- the sharpness value of the parent vertex
- the sharpness values for each incident parent edge
- the sharpness value of the child vertex

- the sharpness values for each incident child edge

The latter should not be surprising given the dependencies noted above. There are also a few more to consider for future use, e.g. whether the EDGE or VERTEX is manifold or not. In most cases, additional information can be provided to the mask queries (i.e. pre-determined Rules), and most of the child sharpness values are not necessary. The most demanding situation is a fractional crease that decays to zero – in which case all parent and child sharpness values in the neighborhood are required to determine the proper transitional weight.

3.4.2 The MASK interface

Methods dealing with the collections of weights defining a mask are typically parameterized by a MASK template parameter that contains the weights. The set of mask weights is currently divided into vertex-weights, edge-weights and face-weights – consistent with previous usage in OpenSubdiv and providing some useful correlation between the full set of weights and topology. The vertex-weights refer to parent vertices incident the parent component from which a vertex originated, the edge-weights the vertices opposite incident edges of the parent, and the face-weights the center of incident parent faces. Note the latter is NOT in terms of vertices of the parent but potentially vertices in the child originating from faces of the parent. This has been done historically in OpenSubdiv but is finding less use – particularly when it comes to providing greater support for the Loop scheme – and is a point needing attention.

So the mask queries require the following capabilities:

- assign the number of vertex, edge and/or face weights
- retrieve the number of vertex, edge and/or face weights
- assign individual vertex, edge and/or face weights by index
- retrieve individual vertex, edge and/or face weights by index

through a set of methods required of all MASK classes. Since the maximum number of weights is typically known based on the topology, usage within Vtr, Far or Hbr is expected to simply define buffers on the stack. Another option is to utilize pre-allocated tables, partitioned into the three sets of weights on construction of a MASK, and populated by the mask queries.

A potentially useful side-effect of this is that the client can define their weights to be stored in either single or double-precision. With that possibility in mind, care was taken within the mask queries to make use of a declared type in the MASK interface (MASK::Weight) for intermediate calculations. Having support for double-precision masks in Sdc does enable it at higher levels in OpenSubdiv if later desired, and that support is made almost trivial with MASK being generic.

It is important to remember here that these masks are being defined consistent with existing usage within OpenSubdiv: both Hbr and the subdivision tables generated by Far. As noted above, the "face weights" correspond to the centers of incident faces, i.e. vertices on the same level as the vertex for which the mask is being computed, and not relative to vertices in the parent level as with the other sets of weights. It is true that the weights can be translated into a set in terms solely of parent vertices, but in the general case (i.e. Catmark subdivision with non-quads in the base mesh) this requires additional topological association. In general we would need N-3 weights for the N-3 vertices between the two incident edges, where N is the number of vertices of each face (typically 4 even at level 0). Perhaps such a translation method could be provided on the mask class, with an optional indication of the incident face topology for the irregular cases. The Loop scheme does not have "face weights", for a vertex-vertex mask, but for an edge-vertex mask it does require weights associated with the faces incident the edge – either the vertex opposite the edge for each triangle, or its center (which has no other use for Loop).

4 Vtr Overview

4.1 Vectorized Topology Representation (Vtr)

Vtr consists of a suite of classes that collectively provide an intermediate representation of topology that supports efficient refinement.

Vtr is intended for internal use only and is currently accessed through the Far layer by the `Far::TopologyRefiner`, which assembles these Vtr classes to meet the topological and refinement needs of the Far layer. What follows is therefore more intended to provide insite into the underlying architecture than to describe particular usage. For documentation more relevant to direct usage, proceed to the Far section previously noted.

Vtr is vectorized in that its topological data is stored more as a collection of vectors of primitive elements rather than as the faces, vertices and edges that make up many other topological representations. It is essentially a structure-of-arrays (SOA) approach to topology in contrast to the more common array-of-structures pattern found in many other topological representations. Vtr's use of vectors allows it to be fairly efficient in its use of memory and similarly efficient to refine, but the topology is fixed once defined.

Vtr classes are purely topological. They are even more independent of the representation of vertices, faces, etc. than Hbr in that they are not even parameterized by an interface to such components. So the same set of Vtr objects can eventually be used to serve more than one representation of these components. The primary requirement is that a mesh be expressable as an indexable set (i.e. a vector or array) of vertices, edges and faces. The index of a component uniquely identifies it and properties are retrieved by referring to it by index.

It's worth qualifying the term "topological" here and elsewhere – we generally refer to "topology" as "subdivision topology" rather than "mesh topology". A subdivision hierarchy is impacted by the presence of semi-sharp creasing, as the subdivision rules change in response to that creasing. So subdivision topology includes the sharpness values assigned to edges and vertices that affect the semi-sharp creasing.

The two primary classes in Vtr consist of:

- Vtr::Level - a class representing complete vertex topology for a level
- Vtr::Refinement - a class mapping a parent Vtr::Level to a child level

Others exist to represent the following:

- selection and appropriate tagging of components for sparse refinement
- divergence of face-varying topology from the vertex topology
- mapping between face-varying topology at successive levels
- common low-level utilities, e.g. simple array classes

4.2 Vtr::Level

Vtr::Level is a complete topological description of a subdivision level, with the topological relations, sharpness values and component tags all stored in vectors (literally std::vectors, but easily changed via typedefs). There are no classes or objects for the mesh component types (i.e. faces, edges and vertices) but simply an integer index to identify each. It can be viewed as a structure-of-arrays representation of the topology: any property related to a particular component is stored in an array and accessible using the index identifying that component. So with no classes the for the components, its difficult to say what constitutes a "vertex" or a "face": they are each the sum of all the fields scattered amongst the many vectors included.

Level represents a single level of a potential hierarchy and is capable of representing the complete base mesh. There are no members that relate data in one level to any other, either below or above. As such, any Level can be used as the base level for a new subdivision hierarchy (potentially more than one). All relationships between separate levels are maintained in the Vtr::Refinement class.

4.2.1 Topological Relationships

Level requires the definition of and associations between a fixed set of indexable components for all three component types, i.e. an explicit edge list in addition to the expected set of vertices and faces. There are no explicit component objects in the representation, only an integer index (Vtr::Index) identifying each component within the set and data associated with that component in the various vectors.

The topology is stored as six sets of incident relations between the components: two each for the two other component types incident each component type, i.e.:

- for each face, its incident vertices and incident edges
- for each edge, its incident vertices and incident faces
- for each vertex, its incident edges and incident faces

The collection of incidence relations is a vectorized variation of AIF (the "Adjacency and Incidence Framework"). The set of these six incidence relations is not minimal (only four are required, but that set excludes the most desired face-vertex relation) but all six are kept and maintained to facilitate faster refinement. While the sizes of several vectors are directly proportional to the number of vertices, edges or faces to which the data is associated, the sizes of some of the vectors for these relations is more cumulative and so additional vectors of offsets is required (typical of the face-vertex list commonly used as the minimal definition of mesh topology).

Vectors for the sharpness values associated with crease edges and corner vertices are included (and so sized according to the number of edges and vertices), along with additional tags for the components that may be helpful to refinement (i.e. the type of subdivision Rule associated with each vertex).

A Level is really just a container for data in a subdivision level, and so its public methods are primarily to access that data. Modification of the data is protected and only made available to classes that are intended to construct Levels: currently the Far factory class that is responsible for building the base level, and the Vtr::Refinement class that constructs subsequent levels during refinement.

4.2.2 Memory Efficiency

One of the advantages in storing data in what is essentially a structure-of-arrays, rather than the array-of-structures more typical of topological representations, is that we can be more selective about memory usage in some cases. Particularly in the case of uniform refinement, when the data in subsequent levels is typically 4x its predecessor, we can minimize what we either generate or keep around at each level. For instance, if only a face-list is required at the finest level, we only need to generate one of the six topological relations: the vertices incident each face. When we do keep Levels around in memory (as is the case with the `Far::TopologyRefiner`) we do have do have the opportunity to prune what is not strictly necessary after the refinement. Just as with construction, whatever classes are privileged to construct a Level are likely those that will be privileged to prune its contents when needed.

The current implementation of Level is far from optimal though – there are opportunities for improvement. After one level of subdivision, the faces in a Level will be either all quads or tris. Having specializations for these cases and using the more general case in support of N-sided faces for the base level only is one possibility. Levels also allocate dozens of vectors in which to store all data. Since these vectors are of fixed size once created, they could be aggregated by partitioning one or a smaller number of larger block of memory into the desired pieces. The desire to make some of these improvements is part of why Vtr is not directly exposed for public use and instead exposed via Far.

4.3 Vtr::Refinement

While Vtr::Level contains the topology for each subdivision level, Vtr::Refinement is responsible for creating a new level via refinement of an existing one, and for maintaining the relationships between the components in the parent and child levels. So a simplified view of a subdivision hierarchy with Vtr is a set of Levels with a Refinement between each successive pair.

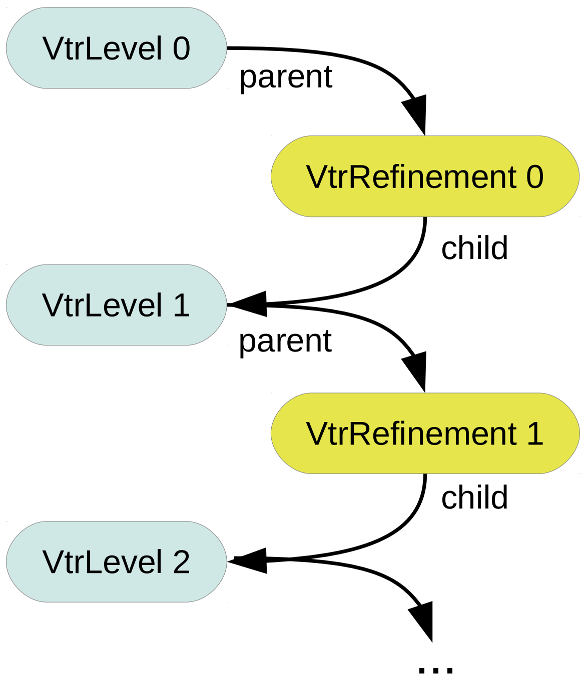


Figure 15: Vtr::Refinement

Refinement is a friend of Level and will populate a child level from a parent given a set of refinement parameters. Aside from parameters related to data or depth, there are two kinds of refinement supported: uniform and sparse. The latter sparse refinement requires selection of an arbitrary set of components – any dependent or “neighboring” components that are required for the limit will be automatically included. So feature-adaptive refinement is just one form of this selective sparse refinement, the criteria being the topological features of interest (creases and extra-ordinary vertices). The intent is to eventually provide more flexibility to facilitate the refinement of particular regions of interest or more dynamic/adaptive needs.

Refinement has also been subclassed according to the type of topological split being performed, i.e. splitting all faces into quads or tris via the QuadRefinement and TriRefinement subclasses. As noted with Vtr::Level, there is further room for improvement in memory and/or performance here by combining more optimal specializations for both Refinement and Level – with consideration of separating the uniform and sparse cases.

4.3.1 Parent-child and child-parent relationships

While Refinement populates a new child Level as part of its refinement operation, it also accumulates the relationships between the parent and child level (and as with Level, this data is stored in vectors indexable by the components).

The associations between components in the two levels was initially only uni-directional: child components were associated with incident components of a parent component based on the parent components topology, so we had a parent-to-child mapping (one to many). Storing the reverse child-to-parent mapping was avoided to reduce memory (particularly in the case of uniform refinement) as it often was not necessary, but a growing need for it, particularly in the case of sparse feature-adaptive refinement, lead to it being included.

4.3.2 Data flexibility

One of the advantages of the structure-of-arrays representation in both Level and Refinement is that we can make more dynamic choices about what type of data we choose to allocate and use based on needs. For instance, we can choose between maintaining the parent-child or child-parent mapping in Refinement, or both if needed, and we can remove one if no longer necessary. An active example of this is uniform refinement: if we only require the face-vertex list at the finest subdivision level, there is no need to generate a complete topological description of that level (as would be required of more traditional representations), and given that level is 4x the magnitude of its parent, the savings are considerable.

Currently there is nothing specific to a subdivision scheme in the refinement other than the type of topological splitting to apply. The refinement does subdivide sharpness values for creasing, but that too is independent of scheme. Tags were added to the base level that are propagated through the refinement and these too are dependent on the scheme, but are applied externally.

5 Far Overview

5.1 Feature Adaptive Representation (Far)

Far is the primary API layer for processing client-supplied mesh data into subdivided surfaces. The Far interface may be used directly and also may be used to prepare mesh data for further processing by Osd. The two main aspects of the subdivision process are Topology Refinement and Primvar Refinement.

5.1.1 Topology Refinement

Topology refinement is the process of splitting the mesh topology according to the specified subdivison rules to generate new topological vertices, edges, and faces. This process is purely topological and does not depend on the speciific values of any primvar data (point positions, etc). Topology refinement can be either uniform or adaptive, where extraordinary features are automatically isolated (see feature adaptive subdivision). The Far topology classes present a public interface for the refinement functionality provided in Vtr, The main classes in Far related to topology refinement are:

Table 9: Topology refinement classes

`TopologyRefiner`	A class encapsulating mesh refinement.
`TopologyLevel`	A class representing one level of refinement within a `TopologyRefiner`.
`TopologyRefinerFactory<MESH>`	A factory class template specialized in terms of the application's mesh representation used to construct `TopologyRefiner` instances.

5.1.2 Primvar Refinement

Primvar refinement is the process of computing values for primvar data (points, colors, normals, texture coordinates, etc) by applying weights determined by the specified subdivision rules. There are many advantages gained by distinguishing between topology refinement and primvar interpolation including the ability to apply a single static topological refinement to multiple primvar instances or to different animated primvar time samples. Far supports methods to refine primvar data at the locations of topological vertices and at arbitrary locations on the subdivision limit surface. The main classes in Far related to primvar refinement are:

Table 10: Primvar refinement classes

PrimvarRefiner	A class implementing refinement of primvar data at the locations of topological vertices.
PatchTable	A representation of the refined surface topology that can be used for efficient evaluation of primvar data at arbitrary locations.
StencilTable	A representation of refinement weights suitable for efficient parallel processing of primvar refinement.
LimitStencilTable	A representation of refinement weights suitable for efficient parallel processing of primvar refinement at arbitrary limit surface locations.

5.2 Far::TopologyRefiner

`TopologyRefiner` is the building block for many other useful classes in Far. It performs refinement of an arbitrary mesh and provides access to the refined mesh topology. It can be used for primvar refinement directly using PrimvarRefiner or indirectly by being used to create a stencil table, or a patch table, etc.

`TopologyRefiner` provides the public refinement methods RefineUniform() and RefineAdapative() which perform refinement operations using Vtr. `TopologyRefiner` provides access to the refined topology via TopologyLevel instances.

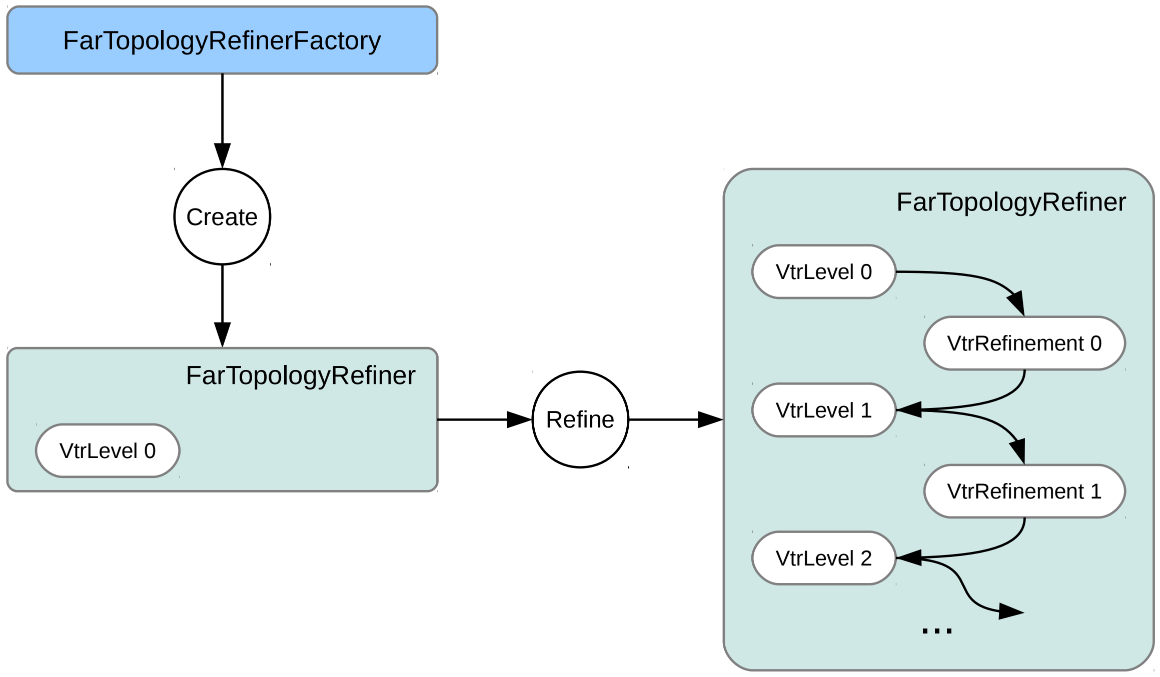


Figure 16: Vtr::TopologyRefiner

5.3 Far::TopologyRefinerFactory

Consistent with other classes in Far, instances of `TopologyRefiner` are created by a factory class – in this case `Far::TopologyRefinerFactory`.

Here we outline several approaches for converting mesh topology into the required `Far::TopologyRefiner`. Additional documentation is provided with the `Far::TopologyRefinerFactory<MESH>` class template used by all, and each has a concrete example provided in one of the tutorials or in the Far code itself.

There are three ways to create TopologyRefiners

- use the existing `TopologyRefinerFactory<TopologyDescriptor>` with a populated instance of `TopologyDescriptor`
- specialize `TopologyRefinerFactory<class MESH>` for more efficient conversion, using only face-vertex information
- fully specialize `TopologyRefinerFactor<class MESH>` for most control over conversion

5.3.1 Use the Far::TopologyDescriptor

Far::TopologyDescriptor is a simple struct that can be initialized to refer to raw mesh topology information – primarily a face-vertex list – and then passed to a provided factory class to create a `TopologyRefiner` from each. Topologically, the minimal requirement consists of:

- the number of vertices and faces of the mesh
- an array containing the number of vertices per face
- an array containing the vertices assigned to each face

These last two define one of the six topological relations that are needed internally by Vtr, but this one relation is sufficient to construct the rest. Additional members are available to assign sharpness values per edge and/or vertex, hole tags to faces, or to define multiple sets (channels) of face-varying data.

Almost all of the Far tutorials (i.e. tutorials/far/tutorial_*) illustrate use of the TopologyDescriptor and its factory for creating TopologyRefiners, i.e. `TopologyRefinerFactory<TopologyDescriptor>`.

For situations when users have raw mesh data and have not yet constructed a boundary representation of their own, it is hoped that this will suffice. Options have even been provided to indicate that raw topology information has been defined in a left-hand winding order and the factory will handle the conversion to right-hand (counter-clockwise) winding on-the-fly to avoid unnecessary data duplication.

5.3.2 Custom Factory for Face Vertices

If the nature of the TopologyDescriptor's data expectations is not helpful, and so conversion to large temporary arrays would be necessary to properly make use of it, it may be worth writing a custom factory.

Specialization of `TopologyRefinerFactory<class MESH>` should be done with care as the goal here is to maximize the performance of the conversion and so minimize overhead due to runtime validation. The template provides the high-level construction of the required topology vectors of the underlying Vtr.

There are two ways to write such a factory: provide only the face-vertex information for topology and let the factory infer all edges and other relationships, or provide the complete edge list and all other topological relationships directly. The latter is considerably more involved and described in a following section.

The definition of `TopologyRefinerFactory<TopologyDescriptor>` provides a clear and complete example of constructing a `TopologyRefiner` with minimal topology information, i.e. the face-vertex list. The class template `TopologyRefinerFactory<MESH>` documents the needs here and the TopologyDescriptor instantiation and specialization should illustrate that.

5.3.3 Custom Factory for Direct Conversion

Fully specializing a factory for direct conversion is needed only for those requiring ultimate control and is not generally recommended. It is recommended that one of the previous two methods initially be used to convert your mesh topology into a `TopologyRefiner`. If the conversion performance is critical, or significant enough to warrant improvement, then it is worth writing a factory for full topological conversion.

Writing a custom factory requires the specification/specialization of two methods with the following purpose:

- specify the sizes of topological data so that vectors can be pre-allocated
- assign the topological data to the newly allocated vectors

As noted above, the assumption here is that the client's boundary-rep knows best how to retrieve the data that we require most efficiently. After the factory class gathers sizing information and allocates appropriate memory, the factory provides the client with locations of the appropriate tables to be populated (using the same Array classes and interface used to access the tables). The client is expected to load a complete topological description along with additional optional data, i.e.:

- the six topological relations required by Vtr, oriented when manifold

- sharpness values for edges and/or vertices (optional)
- additional tags related to the components, e.g. holes (optional)
- values-per-face for face-varying channels (optional) e

This approach requires dealing directly with edges, unlike the other two. In order to convert edges into a TopologyRefiner's representation, the edges need to be expressed as a collection of known size N – each of which is referred to directly by indices [0,N-1]. This can be awkward for representations such as half-edge or quad-edge that do not treat the instance of an edge uniquely.

Particular care is also necessary when representing non-manifold features. The previous two approaches will construct non-manifold features as required from the face-vertex list – dealing with degenerate edges and other non-manifold features as encountered. When directly translating full topology it is necessary to tag non-manifold features, and also to ensure that certain edge relationships are satisfied in their presence. More details are available with the assembly methods of the factory class template.

While there is plenty of opportunity for user error here, that is no different from any other conversion process. Given that Far controls the construction process through the Factory class, we do have ample opportunity to insert runtime validation, and to vary that level of validation at any time on an instance of the Factory. The factory does provide run-time validation on the topology constructed that can be used for debugging purposes.

A common base class has been created for the factory class, i.e.:

```
template <class MESH>
class TopologyRefinerFactory : public TopologyRefinerFactoryBase
```

both to provide common code independent of `<MESH>` and also potentially to protect core code from unwanted specialization.

5.4 Far::PrimvarRefiner

PrimvarRefiner supports refinement of arbitrary primvar data at the locations of topological vertices. A PrimvarRefiner accesses topology data directly from a `TopologyRefiner`.

Different methods are provided to support three different classes of primvar interpolation. These methods may be used to refine primvar data to a specified refinement level.

Interpolate(...)	Interpolate using vertex weights
InterpolateVarying(...)	Interpolate using linear weights
InterpolateFaceVarying(...)	Interpolate using face-varying weights

Additional methods allow primvar data to be interpolated to the final limit surface including the calculation of first derivative tangents.

Limit(dst)	Interpolate to the limit surface using vertex weights
Limit(dst, dstTan1, dstTan2)	Interpolate including first derivatives to the limit surface using vertex weights
LimitFaceVarying(...)	Interpolate to the limit surface using face-varying weights

PrimarRefiner provides a straightforward interface for refining primvar data, but depending on the application use case, it can be more efficient to create and use a StencilTable, or PatchTable, to refine primvar data.

5.5 Far::PatchTable

The patch table is a serialized topology representation. This container is generated using `Far::PatchTableFactory` from an instance `Far::TopologyRefiner` after a refinement has been applied. The `Far::PatchTableFactory` traverses the data-structures of the `TopologyRefiner` and serializes the sub-faces into collections of bi-linear and bi-cubic patches as dictated by the refinement mode (uniform or adaptive). The patches are then sorted into arrays based on their types.

5.5.1 Patch Arrays

The patch table is a collection of control vertex indices. Meshes are decomposed into a collection of patches, which can be of different types. Each type has different requirements for the internal organization of its control-vertices. A PatchArray contains a sequence of multiple patches that share a common set of attributes.

While all patches in a PatchArray will have the same type, each patch in the array is associated with a distinct PatchParam which specifies additional information about the individual patch.

Each PatchArray contains a patch Descriptor that provides the fundamental description of the patches in the array.

The PatchArray ArrayRange provides the indices necessary to track the records of individual patches in the table.

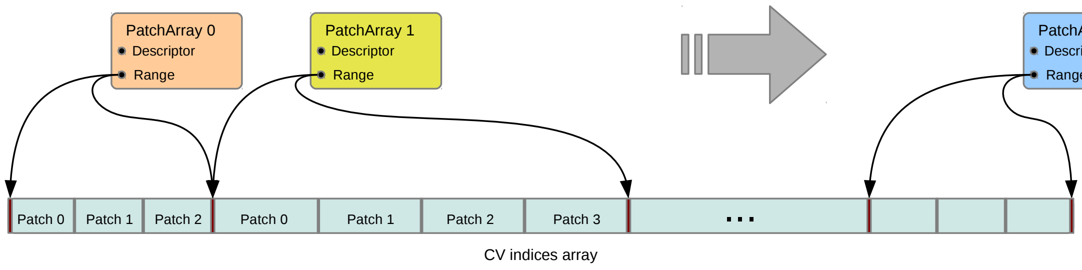


Figure 17: Far::PatchTables

5.5.2 Patch Types

The following are the different patch types that can be represented in the PatchTable:

Patch Type	**#CVs**	**Description**
NON_PATCH	n/a	"Undefined" patch type
POINTS	1	Points : useful for cage drawing
LINES	2	Lines : useful for cage drawing
QUADS	4	Bi-linear quads-only patches
TRIANGLES	3	Bi-linear triangles-only mesh
LOOP	n/a	Loop patch (currently unsupported)
REGULAR	16	B-spline Basis patches
GREGORY	4	Legacy Gregory patches
GREGORY_BOUNDARY	4	Legacy Gregory Boundary patches
GREGORY_BASIS	20	Gregory Basis patche

The type of a patch dictates the number of control vertices expected in the table as well as the method used to evaluate values.

5.5.3 Patch Parameterization

Each patch represents a specific portion of the parametric space of the coarse topological face identified by the PatchParam FaceId. As topological refinement progresses through successive levels, each resulting patch corresponds to a smaller and smaller subdomain of the face. The PatchParam UV origin describes the mapping from the uv domain of the patch to the uv subdomain of the topological face. We encode this uv origin using log2 integer values for compactness and efficiency.

It is important to note that this uv parameterization is the intrinsic parameterization within a given patch or coarse face and is distinct from any client specified face-varying channel data.

Patches which result from irregular coarse faces (non-quad faces in the Catmark scheme, or non-trianglular faces in the Loop scheme) are offset by the one additional level needed to "quadrangulate" or "triangulate" the irregular face.

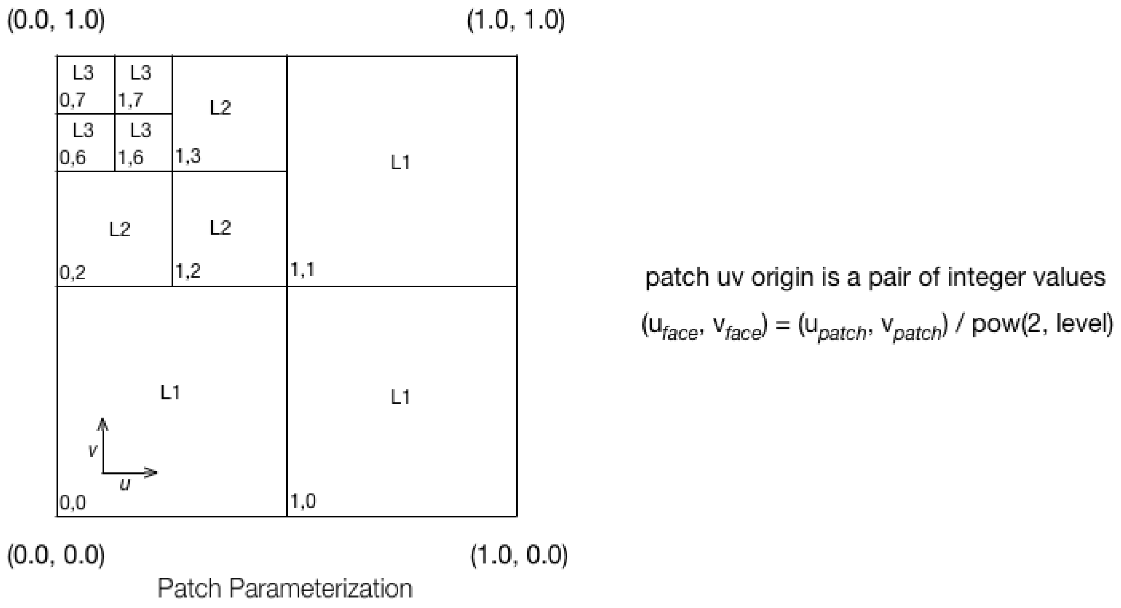


Figure 18: Far patch parameter

A patch along an interpolated boundary edge is supported by an incomplete sets of control vertices. For consistency, patches in the PatchTable always have a full set of control vertex indices and the PatchParam Boundary bitmask identifies which control vertices are incomplete (the incomplete control vertex indices are assigned values which duplicate the first valid index). Each bit in the boundary bitmask corresponds to one edge of the patch starting from the edge from the first vertex and continuing around the patch. With feature adaptive refinement, regular B-spline basis patches along interpolated boundaries will fall into one of the eight cases (four boundary and four corner) illustrated below:

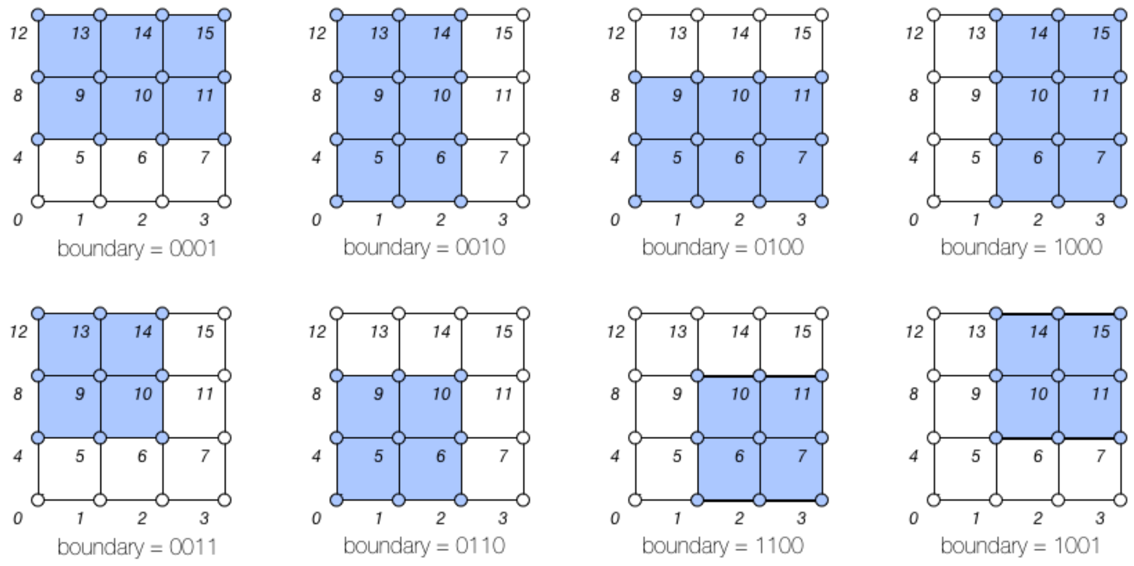


Figure 19: Far patch boundary

Transition edges occur during feature adaptive refinement where a patch at one level of refinement is adjacent to pairs of patches at the next level of refinement. These T-junctions do not pose a problem when evaluating primvar data on patches, but they must be taken into consideration when tessellating patches (e.g. while drawing) in order to avoid cracks. The PatchParam Transition bitmask identifies the transition edges of a patch. Each bit in the bitmask corresponds to one edge of the patch just like the encoding of boundary edges.

After refining an arbitrary mesh, any of the 16 possible transition edge configurations might occur. The method of handling transition edges is delegated to patch drawing code.

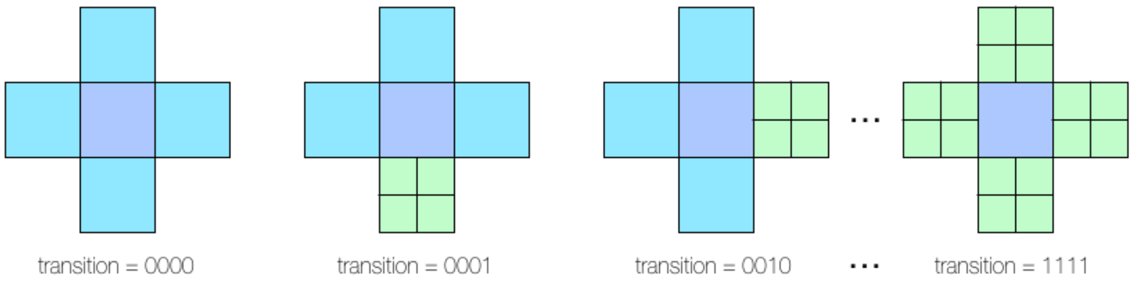


Figure 20: Far patch transition

5.5.4 Single-Crease patches

Using single-crease patches allows a mesh with creases to be represented with many fewer patches than would be needed otherwise. A single-crease patch is a variation of a regular BSpline patch with one additional crease sharpness parameter.

5.5.5 Local Points

The control vertices represented by a PatchTable are primarily refined points, i.e. points which result from applying the subdivision scheme uniformly or adaptively to the points of the coarse mesh. However, the final patches generated from irregular faces, e.g. patches incident on an extraordinary vertex might have a representation which requires additional local points.

5.5.6 Legacy Gregory Patches

Using Gregory patches to approximate the surface at the final patches generated from irregular faces is an alternative representation which does not require any additional local points to be computed. Instead, when Legacy Gregory patches are used, the PatchTable must also have an alternative representation of the mesh topology encoded as a vertex valence table and a quad offsets table.

5.6 Far::StencilTable

The base container for stencil data is the StencilTable class. As with most other Far entities, it has an associated `StencilTableFactory` that requires a `TopologyRefiner`.

5.6.1 Advantages

Stencils are used to factorize the interpolation calculations that subdivision schema apply to vertices of smooth surfaces. If the topology being subdivided remains constant, factorizing the subdivision weights into stencils during a pre-compute pass yields substantial amortizations at run-time when re-posing the control cage.

Factorizing the subdivision weights also allows to express each subdivided vertex as a weighted sum of vertices from the control cage. This step effectively removes any data inter-dependency between subdivided vertices : the computations of subdivision interpolation can be applied to each vertex in parallel without any barriers or constraint. The Osd classes leverage these properties by exploiting CPU and GPU parallelism.

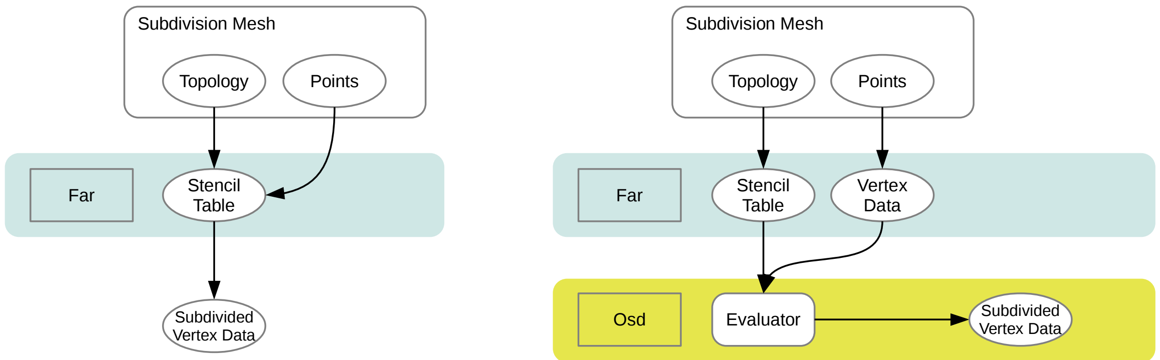


Figure 21: Far stencil table

5.6.2 Principles

Iterative subdivision algorithms converge towards the limit surface by successively refining the vertices of the coarse control cage. Each successive iteration interpolates the new vertices by applying polynomial weights to a basis of supporting vertices.

The interpolation calculations for any given vertex can be broken down into sequences of multiply-add operations applied to the supporting vertices. Stencil table encodes a factorization of these weighted sums : each stencils is created by combining the list of control vertices from the 1-ring.

With iterative subdivision, each refinement step is dependent upon the previous subdivision step being completed, and a substantial number of steps may be required in order approximate the limit : each subdivision step incurs an O(4n) growing amount of computations.

Instead, once the weights of the contributing coarse control vertices for a given refined vertex have been factorized, it is possible to apply the stencil and directly obtain the interpolated vertex data without having to process the data for the intermediate refinement levels.

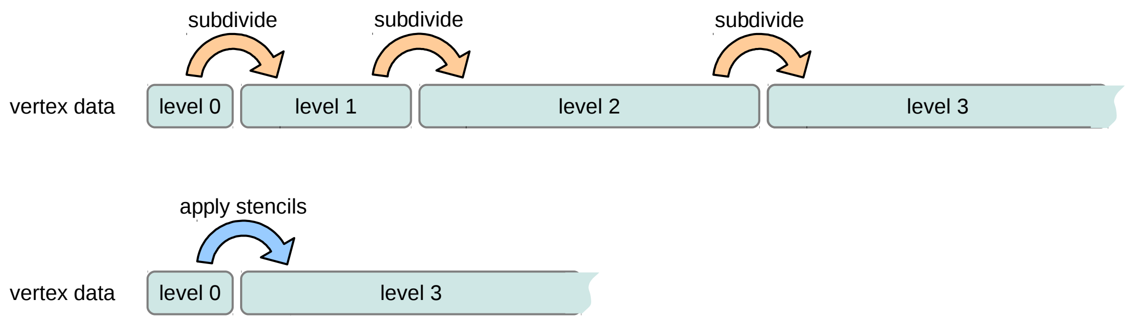


Figure 22: Far stencil table

5.6.3 Cascading Stencils

Client-code can control the amount of factorization of the stencils : the tables can be generated with contributions all the way from a basis of coarse vertices, or reduced only to contributions from vertices from the previous level of refinement.

The latter mode allows client-code to access and insert modifications to the vertex data at set refinement levels (see hierarchical vertex edits). Once the edits have been applied by the client-code, another set of stencils can be used to smoothe the vertex data to a higher level of refinement.

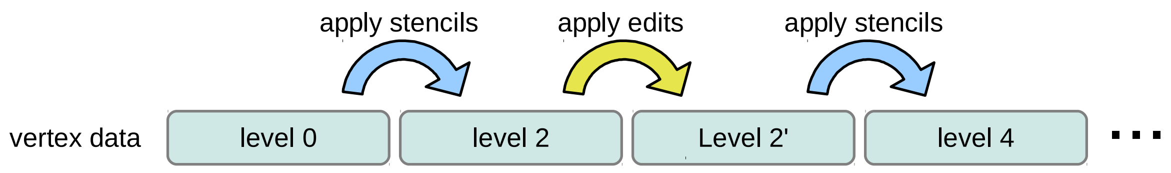


Figure 23: Far stencil table

See implementation details, see the Far cascading stencil tutorial.

5.7 Limit Stencils

Stencil tables can be trivially extended from discrete subdivided vertices to arbitrary locations on the limit surface. Aside from extraordinary points, every location on the limit surface can be expressed as a closed-form weighted average of a set of coarse control vertices from the 1-ring surrounding the face.

The weight accumulation process is similar : the control cage is adaptively subdivided around extraordinary locations. A stencil is then generated for each limit location simply by factorizing the bi-cubic Bspline patch weights over those of the contributing basis of control-vertices.

The use of bi-cubic patches also allows the accumulation of analytical derivatives, so limit stencils carry a set of weights for tangent vectors.

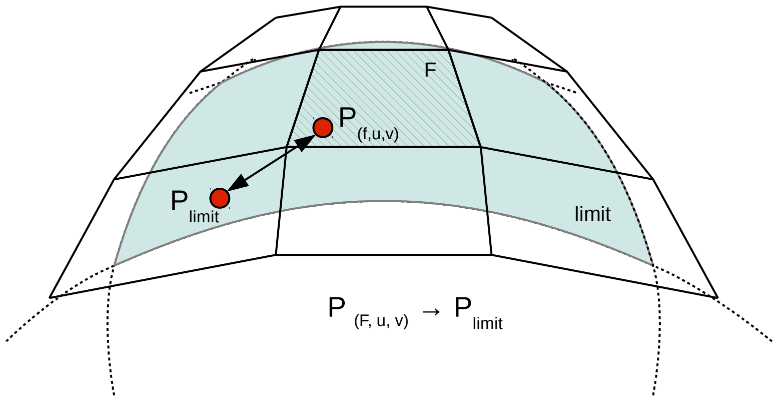


Figure 24: Far limit stencil

Once the stencil table has been generated, limit stencils are the most direct and efficient method of evaluation of specific locations on the limit of a subdivision surface, starting from the coarse vertices of the control cage.

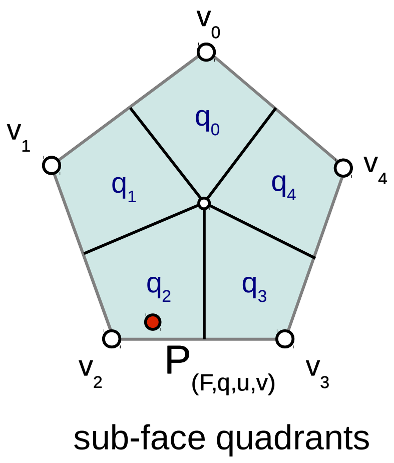
Also: just as discrete stencils, limit stencils that are factorized from coarse control vertices do not have inter-dependencies and can be evaluated in parallel.

For implementation details, see the glStencilViewer code example.

5.7.1 Sample Location On Extraordinary Faces

Each stencil is associated with a singular parametric location on the coarse mesh. The parametric location is defined as face location and local [0.0 - 1.0] (u,v) triplet:

In the case of face that are not quads, a parametric sub-face quadrant needs to be identified. This can be done either explicitly or implicitly by using the unique ptex face indices for instance.



5.7.2 Code Example

When the control vertices (controlPoints) move in space, the limit locations can be very efficiently recomputed simply by applying the blending weights to the series of coarse control vertices:

```
class StencilType {
public:

    void Clear() {
        memset( &x, 0, sizeof(StencilType));
    }

    void AddWithWeight( StencilType const & cv, float weight  ) {
        x += cv.x * weight;
        y += cv.y * weight;
        z += cv.z * weight;
    }

    float x,y,z;
};

std::vector<StencilType> controlPoints,
                         points,
                         utan,
                         vtan;

// Update points by applying stencils
controlStencils.UpdateValues<StencilType>( &controlPoints[0], &points[0] );

// Update tangents by applying derivative stencils
controlStencils.UpdateDerivs<StencilType>( &controlPoints[0], &utan[0], &vtan[0] );
```

6 Osd overview

6.1 OpenSubdiv cross platform (Osd)

Osd contains device dependent code that makes Far structures available on various backends such as TBB, CUDA, OpenCL, GLSL, etc. The main roles of Osd are:

Refinement
Compute stencil-based uniform/adaptive subdivision on CPU/GPU backends

Limit Stencil Evaluation
Compute limit surfaces by limit stencils on CPU/GPU backends

Limit Evaluation with PatchTable
Compute limit surfaces by patch evaluation on CPU/GPU backends

OpenGL/DX11 Drawing with hardware tessellation
Provide GLSL/HLSL tessellation functions for patch table

Interleaved/Batched buffer configuration
Provide consistent buffer descriptor to deal with arbitrary buffer layout.

Cross-Platform Implementation
Provide convenient classes to interop between compute and draw APIs

These are independently used by clients. For example, a client can use only the limit stencil evaluation, or a client can refine subdivision surfaces and draw them with the PatchTable and Osd tessellation shaders. All device specific evaluation kernels are implemented in the Evaluator classes. Since Evaluators don't own vertex buffers, clients should provide their own buffers as a source and destination. There are some interop classes defined in Osd for convenience.

OpenSubdiv utilizes a series of regression tests to compare and enforce identical results across different computational devices.

6.2 Refinement

Osd supports both uniform subdivision and adaptive subdivision.

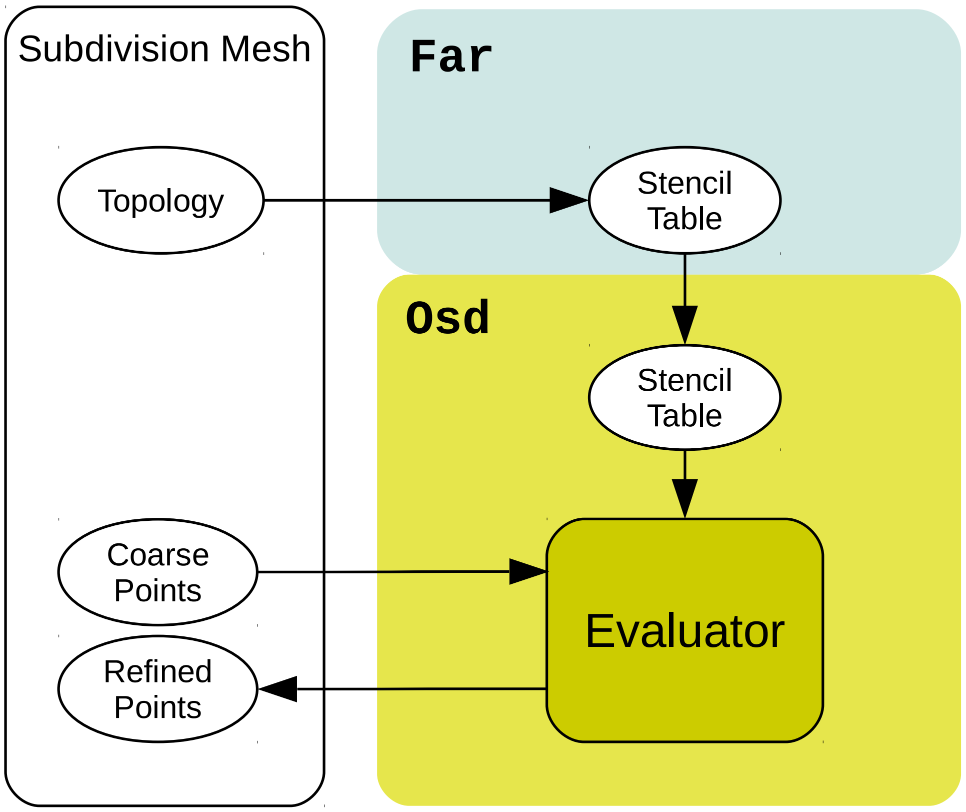


Figure 25: Osd refinement

Once clients create a Far::StencilTable for the topology, they can convert it into device-specific stencil tables if necessary. The following table shows which evaluator classes and stencil table interfaces can be used together. Note that while Osd provides these stencil table classes which can be easily constructed from Far::StencilTable, clients aren't required to use these table classes. Clients may have their own entities as a stencil tables as long as Evaluator::EvalStencils() can access the necessary interfaces.

Backend	Evaluator class	Compatible stencil table
CPU (CPU single-threaded)	CpuEvaluator	Far::StencilTable
TBB (CPU multi-threaded)	TbbEvaluator	Far::StencilTable
OpenMP (CPU multi-threaded)	OmpEvaluator	Far::StencilTable
CUDA (GPU)	CudaEvaluator	CudaStencilTable
OpenCL (CPU/GPU)	CLEvaluator	CLStencilTable
GL ComputeShader (GPU)	GLComputeEvaluator	GLStencilTableSSBO
GL Transform Feedback (GPU)	GLXFBEvaluator	GLStencilTableTBO
DX11 ComputeShader (GPU)	D3D11ComputeEvaluator	D3D11StencilTable

6.3 Limit Stencil Evaluation

Limit stencil evaluation is quite similar to refinement in Osd. At first clients create Far::LimitStencilTable for the locations to evaluate the limit surfaces, then convert it into an evaluator compatible stencil table and call Evaluator::EvalStencils().

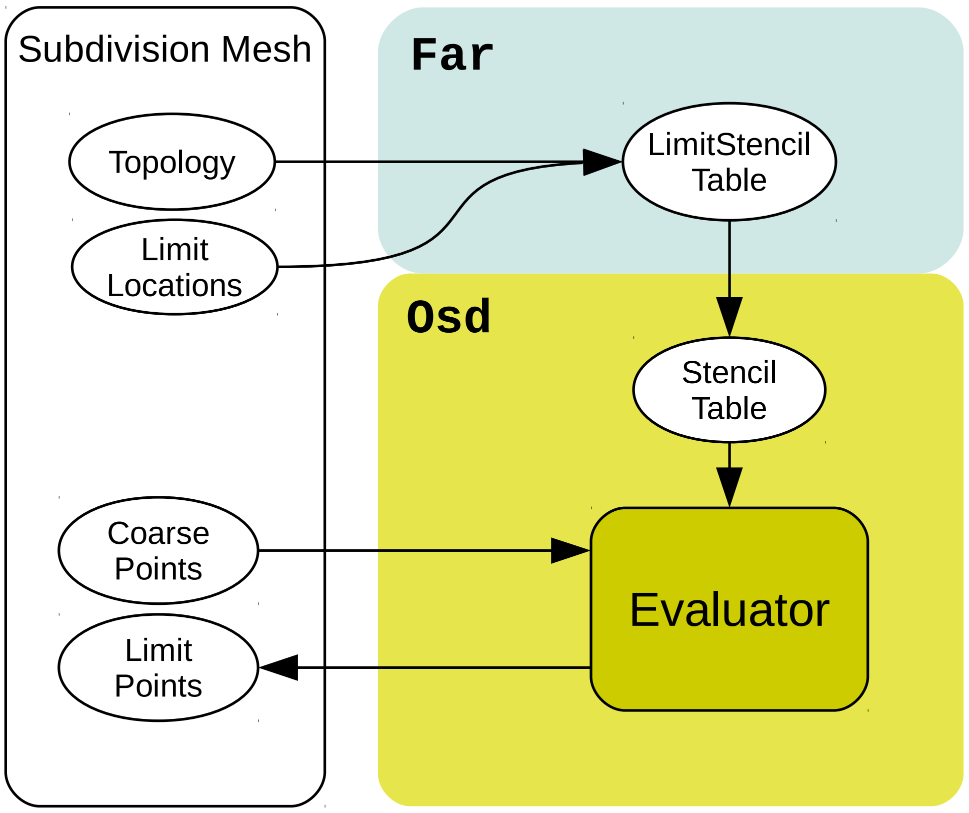


Figure 26: Osd limit stencil evaluation

6.4 Limit Evaluation with PatchTable

Another way to evaluate the limit surfaces is to use the PatchTable. Once all control vertices and local points are resolved by the stencil evaluation, Osd can evaluate the limit surfaces through the PatchTable.

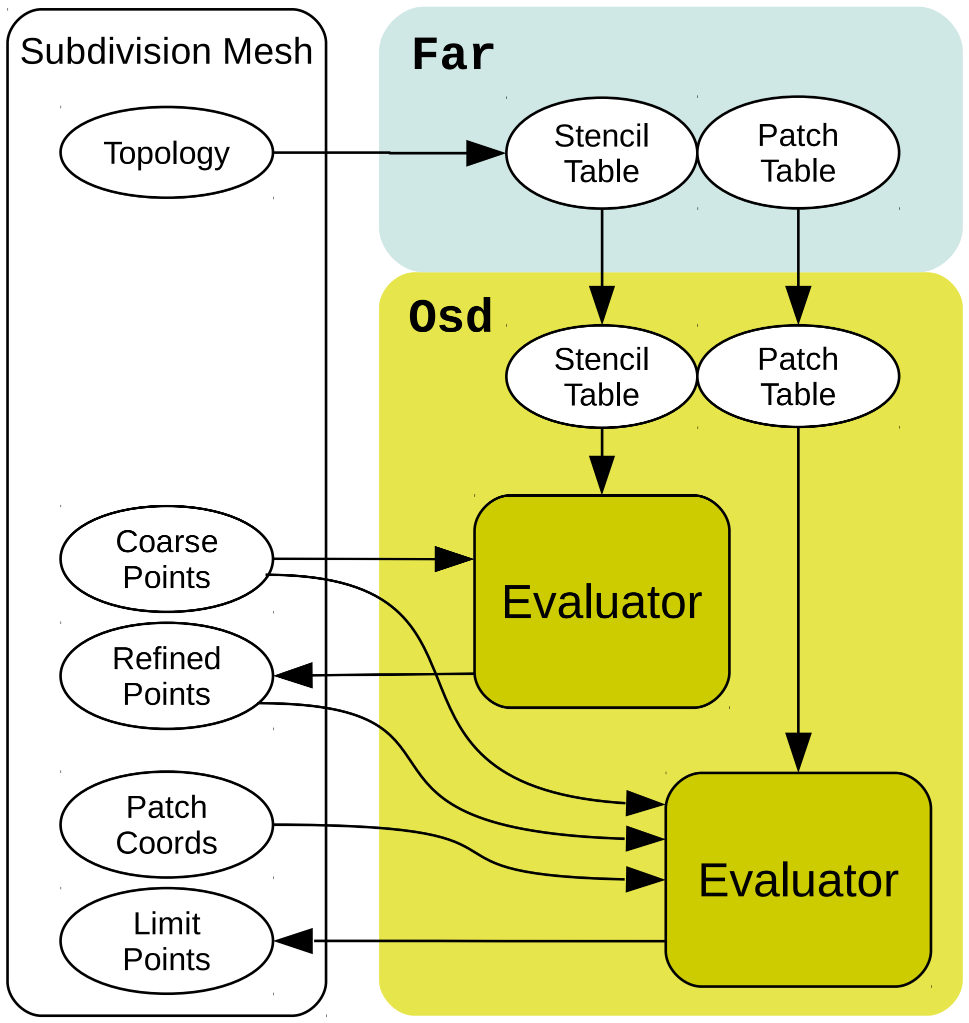


Figure 27: Osd limit evaluation with PatchTable

Backend	**Evaluator class**	**Compatible patch table**
CPU (CPU single-threaded)	CpuEvaluator	CpuPatchTable
TBB (CPU multi-threaded)	TbbEvaluator	CpuPatchTable
OpenMP (CPU multi-threaded)	OmpEvaluator	CpuPatchTable
CUDA (GPU)	CudaEvaluator	CudaPatchTable
OpenCL (CPU/GPU)	CLEvaluator	CLPatchTable
GL ComputeShader (GPU)	GLComputeEvaluator	GLPatchTable
GL Transform Feedback (GPU)	GLXFBEvaluator	GLPatchTable
DX11 ComputeShader (GPU)	D3D11ComputeEvaluator (*)not yet supported	D3D11PatchTable

6.5 OpenGL/DX11 Drawing with Hardware Tessellation

One of the most interesting use cases of the Osd layer is realtime drawing of subdivision surfaces using hardware tessellation. This is somewhat similar to limit evaluation with PatchTable described above.

Drawing differs from limit evaluation in that Osd provides shader snippets for patch evaluation and clients will inject them into their own shader source.

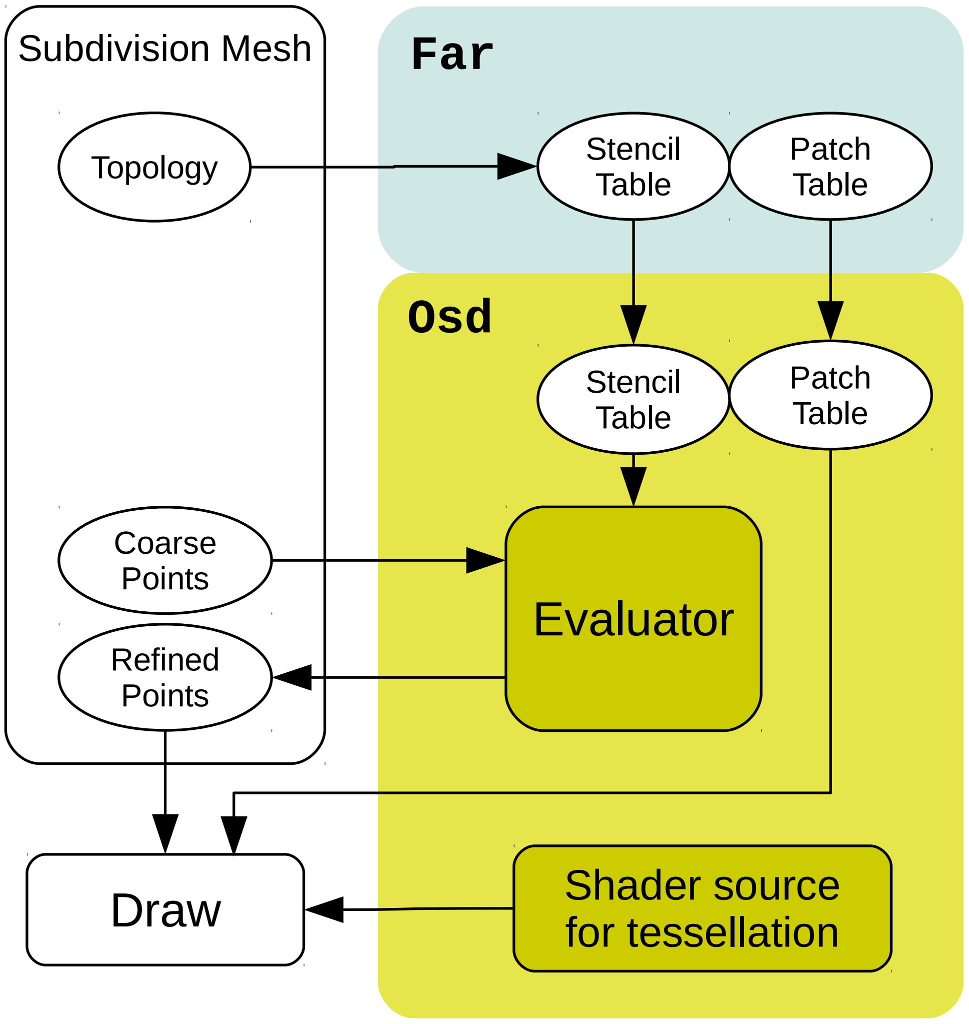


Figure 28: Osd tessellation

See shader interface section for a more detailed discussion of the shader interface.

6.6 Interleaved/Batched Buffer Configuration

All Osd layer APIs assume that each primitive variables to be computed (points, colors, uvs ...) are contiguous arrays of 32bit floating point values. The Osd API refers to such an array as a “buffer”. A buffer can exist on CPU memory or GPU memory. Osd Evaluators typically take one source buffer and one destination buffer, or three destination buffers if derivatives are being computed. Osd Evaluators also take BufferDescriptors, that are used to specify the layout of the source and destination buffers. A BufferDescriptor is a struct of 3 integers which specify an offset, length and stride.

For example:

Vertex 0			Vertex 1			...
X	Y	Z	X	Y	Z	...

The layout of this buffer can be described as

```
Osd::BufferDescriptor desc(/*offset = */ 0, /*length = */ 3, /*stride = */ 3);
```

BufferDescriptor can also be used for an interleaved buffer.

Vertex 0							Vertex 1							...
X	Y	Z	R	G	B	A	X	Y	Z	R	G	B	A	...

```
Osd::BufferDescriptor xyzDesc(0, 3, 7);
Osd::BufferDescriptor rgbaDesc(3, 4, 7);
```

Although the source and destination buffers don't need to be the same buffer for EvalStencils(), adaptive patch tables are constructed to first index the coarse vertices and the refined vertices immediately afterward. In this case, the BufferDescriptor for the destination should include the offset as the number of coarse vertices to be skipped.

Coarse vertices (n) : Src							Refined vertices : Dst						
Vertex 0			Vertex 1			...	Vertex n			Vertex n+1			...
X	Y	Z	X	Y	Z	...	X	Y	Z	X	Y	Z	...

```
Osd::BufferDescriptor srcDesc(0, 3, 3);
Osd::BufferDescriptor dstDesc(n*3, 3, 3);
```

Also note that the source descriptor doesn't have to start with offset = 0. This is useful when a client has a big buffer with multiple objects batched together.

6.7 Cross-Platform Implementation

One of the key goals of OpenSubdiv is to achieve as much cross-platform flexibility as possible and leverage all optimized hardware paths where available. This can be very challenging as there is a very large variety of plaftorms and APIs available, with very distinct capabilities.

In Osd, Evaluators don't care about interops between those APIs. All Evaluators have two kinds of APIs for both EvalStencils() and EvalPatches().

- Explicit signatures which directly take device-specific buffer representation (e.g., pointer for CpuEvaluator, GLuint buffer for GLComputeEvaluator, etc.)
- Generic signatures which take arbitrary buffer classes. The buffer class is required to have a certain method to return the device-specific buffer representation.

The later interface is useful if the client supports multiple backends at the same time. The methods that need to be implemented for the Evaluators are:

Evaluator class	object	method
CpuEvaluator TbbEvaluator OmpEvaluator	pointer to cpu memory	BindCpuBuffer()
CudaEvaluator	pointer to cuda memory	BindCudaBuffer()
CLEvaluator	cl_mem	BindCLBuffer()
GLComputeEvaluator GLXFBEvaluator	GL buffer object	BindVBO()
D3D11ComputeEvaluator	D3D11 UAV	BindD3D11UAV()

The buffers can use these methods as a trigger of interop. Osd provides a default implementation of interop buffer for most of the backend combinations. For example, if the client wants to use CUDA as a computation backend and use OpenGL as the drawing API, Osd::CudaGLVertexBuffer fits the case since it implements BindCudaBuffer() and BindVBO(). Again, clients can implement their own buffer class and pass it to the Evaluators.

7 Osd Tesslation shader interface

7.1 Basic

Starting with 3.0, Osd tessellation shaders can be used as a set of functions from client shader code. In order to tessellate Osd patches, client shader code should perform the following steps (regular B-spline patch case):

In a tessellation control shader

- fetch a PatchParam for the current patch
- call OsdComputePerPatchVertexBSpline() to compute OsdPerPatchVertexBezier.
- compute tessellation level. To prevent cracks on transition patches, two vec4 parameters (tessOuterHi, tessOuterLo) will be needed in addition to built-in gl_TessLevelInner/Outers.

In a tessellation evaluation shader

- call OsdGetTessParameterization() to remap gl_TessCoord to a patch parameter at which to evaluate.
- call OsdEvalPatchBezier()/OsdEvalPatchGregory() to evaluate the current patch.

The following is a minimal example of GLSL code explaining how client shader code uses OpenSubdiv shader functions to tessellate patches of a patch table.

7.1.1 Tessellation Control Shader Example (for BSpline patches)

```
layout (vertices = 16) out;
in vec3 position[];
patch out vec4 tessOuterLo, tessOuterHi;
out OsdPerPatchVertexBezier v;

void main()
{
    // Get a patch param from texture buffer.
    ivec3 patchParam = OsdGetPatchParam(gl_PrimitiveID);

    // Compute per-patch vertices.
    OsdComputePerPatchVertexBSpline(patchParam, gl_InvocationID, position, v);

    // Compute tessellation factors.
    if (gl_InvocationID == 0) {
        vec4 tessLevelOuter = vec4(0);
        vec2 tessLevelInner = vec2(0);
        OsdGetTessLevelsUniform(patchParam,
                                tessLevelOuter, tessLevelInner,
                                tessOuterLo, tessOuterHi);

        gl_TessLevelOuter[0] = tessLevelOuter[0];
        gl_TessLevelOuter[1] = tessLevelOuter[1];
        gl_TessLevelOuter[2] = tessLevelOuter[2];
        gl_TessLevelOuter[3] = tessLevelOuter[3];

        gl_TessLevelInner[0] = tessLevelInner[0];
        gl_TessLevelInner[1] = tessLevelInner[1];
    }
}
```

7.1.2 Tessellation Evaluation Shader Example (for BSpline patches)

```
layout(quads) in;
patch in vec4 tessOuterLo, tessOuterHi;
in OsdPerPatchVertexBezier v[];
uniform mat4 mvpMatrix;

void main()
{
    // Compute tesscoord.
    vec2 UV = OsdGetTessParameterization(gl_TessCoord.xy, tessOuterLo, tessOuterHi);

    vec3 P = vec3(0), dPu = vec3(0), dPv = vec3(0);
    vec3 N = vec3(0), dNu = vec3(0), dNv = vec3(0);
    ivec3 patchParam = inpt[0].v.patchParam;

    // Evaluate patch at the tess coord UV
    OsdEvalPatchBezier(patchParam, UV, v, P, dPu, dPv, N, dNu, dNv);

    // Apply model-view-projection matrix.
    gl_Position = mvpMatrix * vec4(P, 1);
}
```

7.2 Basis conversion

7.2.1 B-Spline Patch

The following diagram shows how the Osd shaders process b-spline patches.

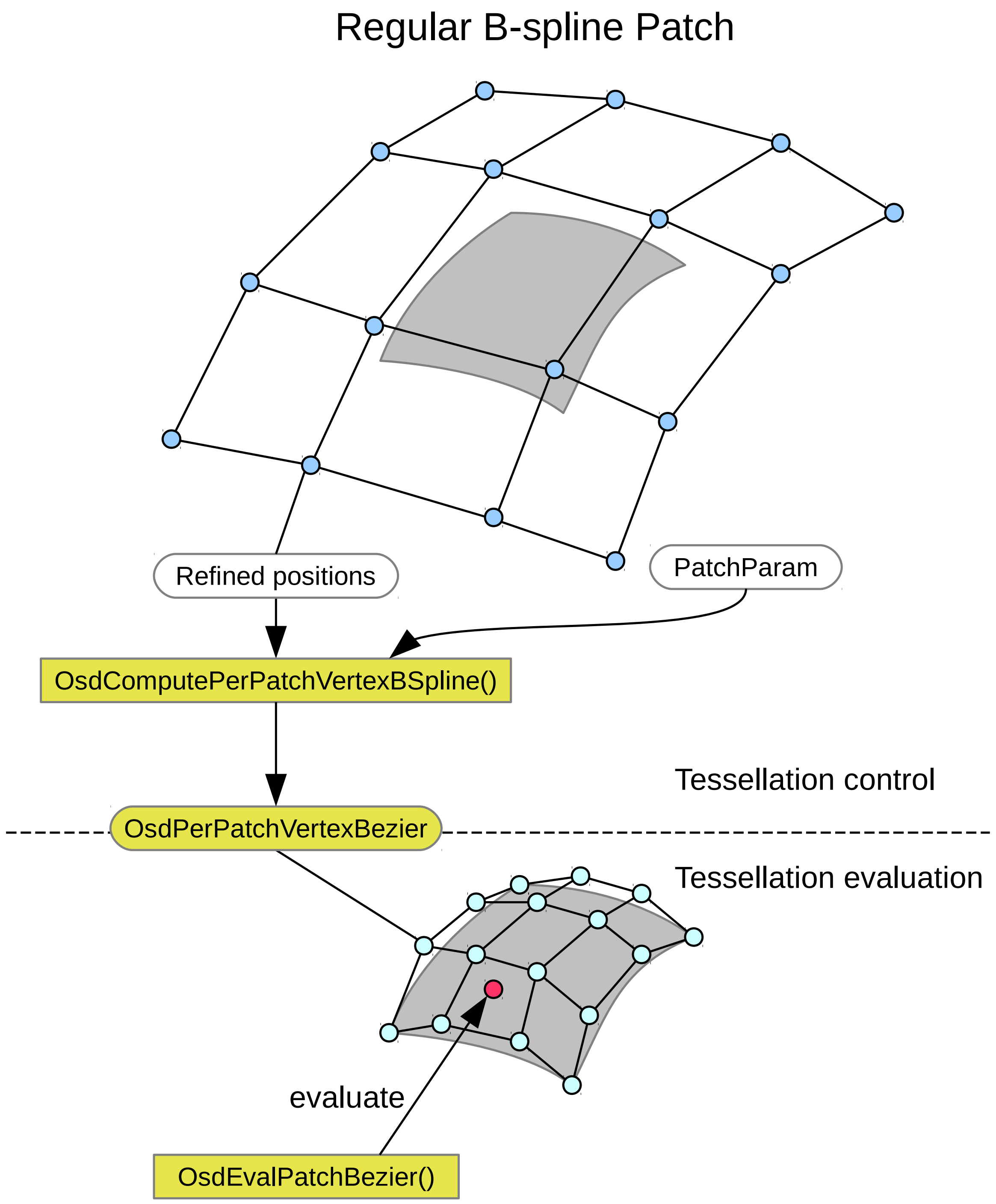


Figure 29: Osd BSpline shader

While regular patches are expressed as b-spline patches in Far::PatchTable, the Osd shader converts them into Bezier basis patches for simplicity and efficiency. This conversion is performed in the tessellation control stage. The boundary edge evaluation and single crease matrix evaluation are also resolved during this conversion. OsdComputePerPatchVertexBSpline() can be used for this process. The resulting Bezier control vertices are stored in OsdPerPatchVertexBezier struct.

```
void  OsdComputePerPatchVertexBSpline(
    ivec3 patchParam, int ID, vec3 cv[16], out OsdPerPatchVertexBezier result);
```

The tessellation evaluation shader takes an array of OsdPerPatchVertexBezier struct, and then evaluates the patch using the OsdEvalPatchBezier() function.

```
void OsdEvalPatchBezier(ivec3 patchParam, vec2 UV,
                        OsdPerPatchVertexBezier cv[16],
                        out vec3 P, out vec3 dPu, out vec3 dPv,
                        out vec3 N, out vec3 dNu, out vec3 dNv)
```

7.2.2 Gregory Basis Patch

In a similar way, Gregory basis patches are processed as follows:

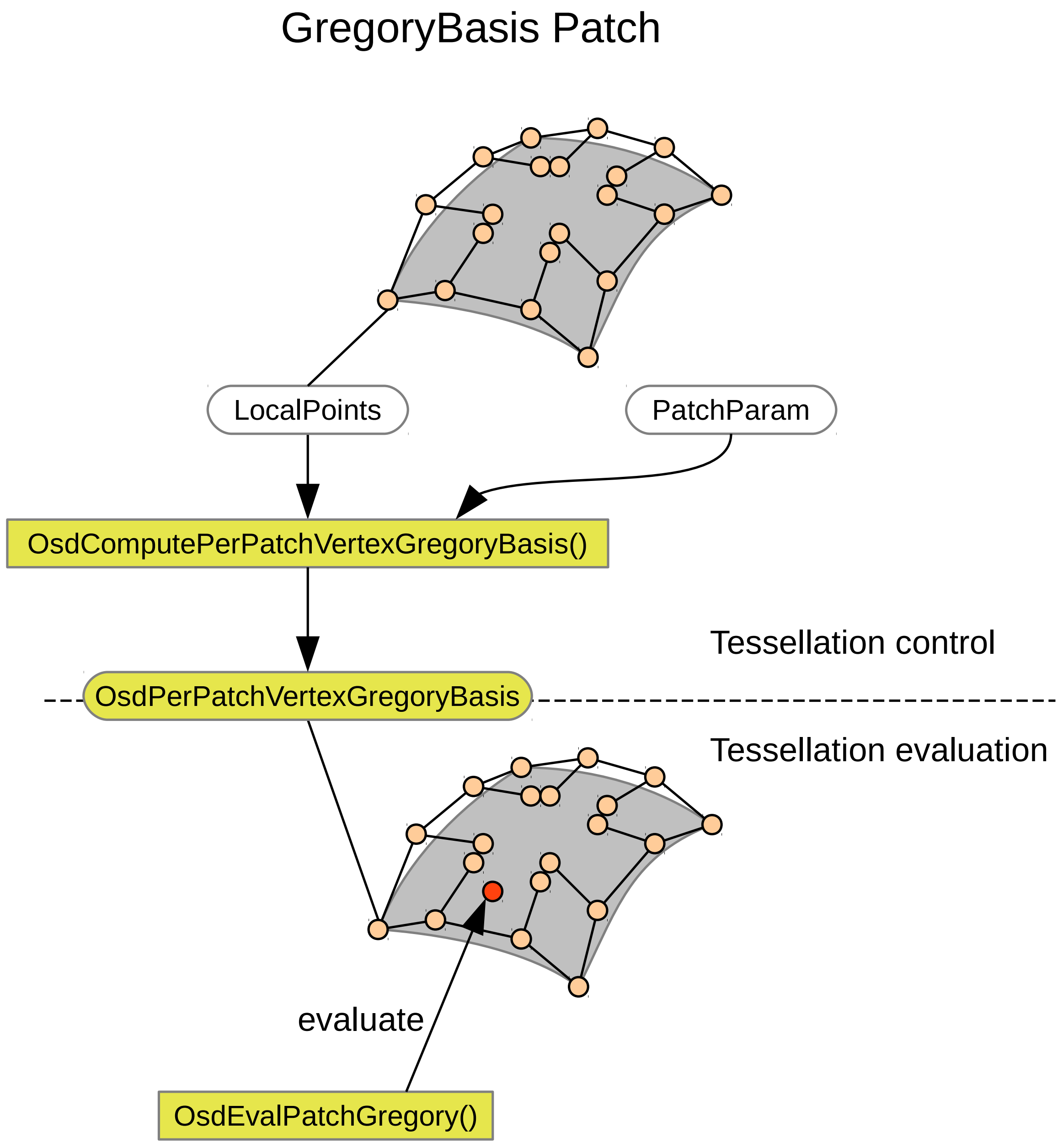


Figure 30: Osd Gregory shader

OsdComputePerPatchVertexGregoryBasis() can be used for the Gregory patches (although no basis conversion involved for the Gregory patches) and the resulting vertices are stored in a OsdPerPatchVertexGreogryBasis struct.

```
void OsdComputePerPatchVertexGregoryBasis(
  ivec3 patchParam, int ID, vec3 cv, out OsdPerPatchVertexGregoryBasis result)
```

The tessellation evaluation shader takes an array of OsdPerPatchVertexGregoryBasis struct, and then evaluates the patch using the OsdEvalPatchGregory() function.

```
void
OsdEvalPatchGregory(ivec3 patchParam, vec2 UV, vec3 cv[20],
                    out vec3 P, out vec3 dPu, out vec3 dPv,
                    out vec3 N, out vec3 dNu, out vec3 dNv)
```

7.2.3 Legacy Gregory Patch (2.x compatibility)

OpenSubdiv 3.0 also supports 2.x style Gregory patch evaluation (see Far overview). In order to evaluate a legacy Gregory patch, client needs to bind extra buffers and to perform extra steps in the vertex shader as shown in the following diagram:

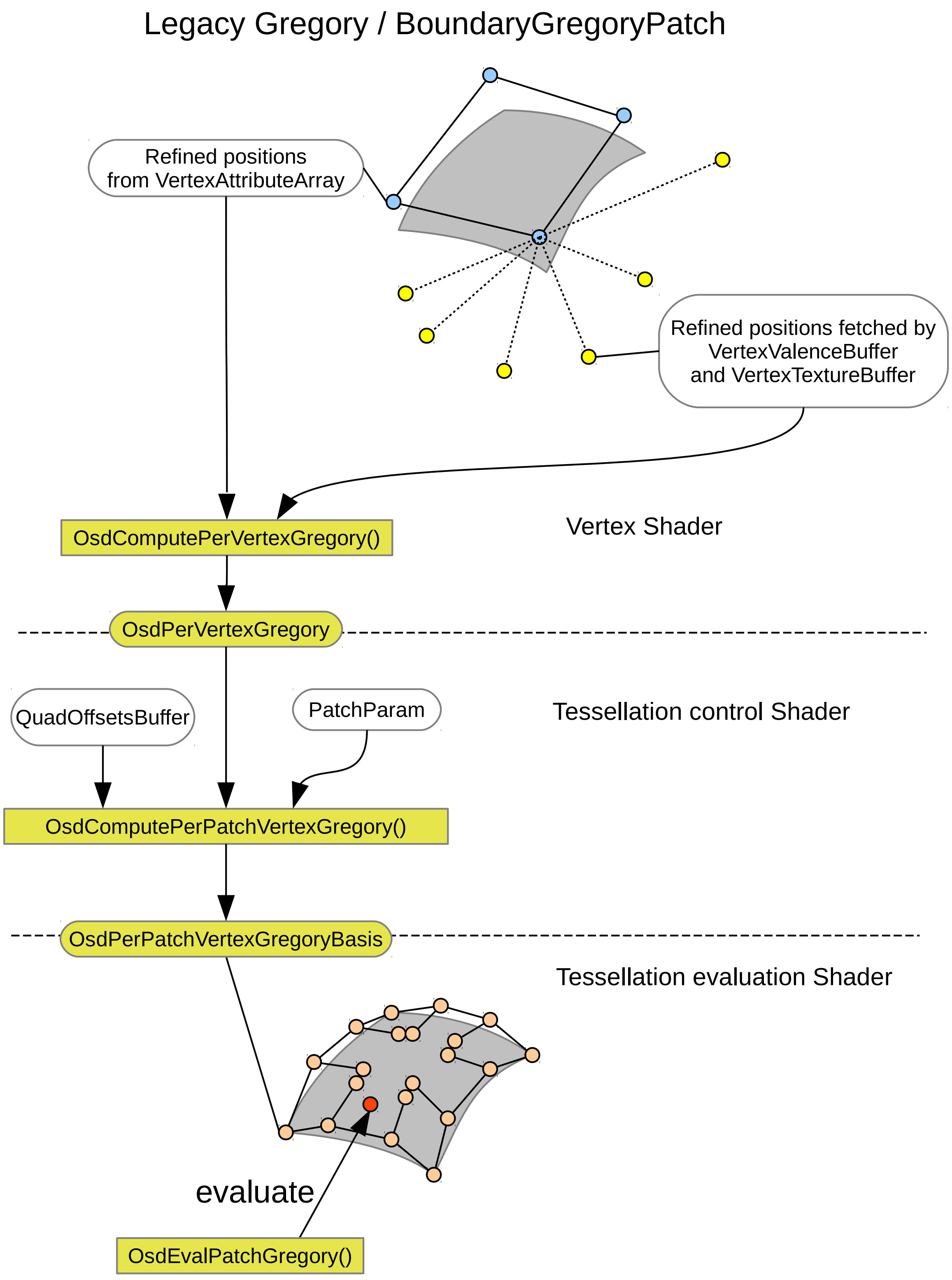

Legacy Gregory / BoundaryGregoryPatch
Refined positions from VertexAttributeArray
Refined positions fetched by VertexValenceBuffer and VertexTextureBuffer
OsdComputePerVertexGregory()
Vertex Shader
OsdPerVertexGregory
QuadOffsetsBuffer
PatchParam
Tessellation control Shader
OsdComputePerPatchVertexGregory()
OsdPerPatchVertexGregoryBasis
Tessellation evaluation Shader
evaluate
OsdEvalPatchGregory()

Figure 31: Osd Legacy gregory shader

7.3 Tessellation levels

Osd provides both uniform and screen-space adaptive tessellation level computation.

Uniform tessellation	OsdGetTessLevelsUniform()
Screen-space adaptive tessellation	OsdGetTessLevelsAdaptiveLimitPoints()

Because of the nature of feature adaptive subdivision, we need to pay extra attention for a patch's outer tessellation level for the screen-space adaptive case so that cracks don't appear.

An edge of the patch marked as a transition edge is split into two segments (Hi and Lo).

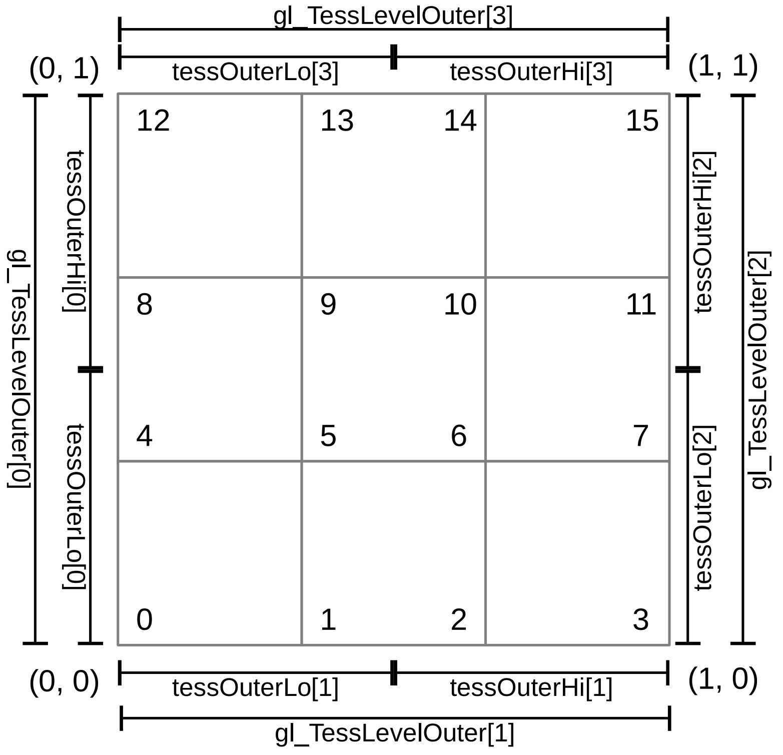


Figure 32: Osd shader patch configuration

The Osd shaders uses these two segments to ensure the same tessellation along the edge between different levels of subdivision. In the following example, suppose the left hand side patch has determined the tessellation level of its right edge to be 5. gl_TessLevelOuter is set to 5 for the edge, and at the same time we also pass 2 and 3 to the tessellation evaluation shader as separate levels for the two segments of the edge split at the middle.

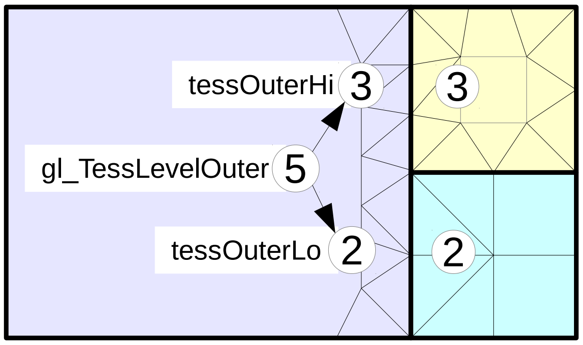


Figure 33: Osd shader transition patch

Then the tessellation evaluation shader takes gl_TessCoord and those two values, and remaps gl_TessCoord using OsdGetTessParameterization() to ensure the parameters are consistent across adjacent patches.

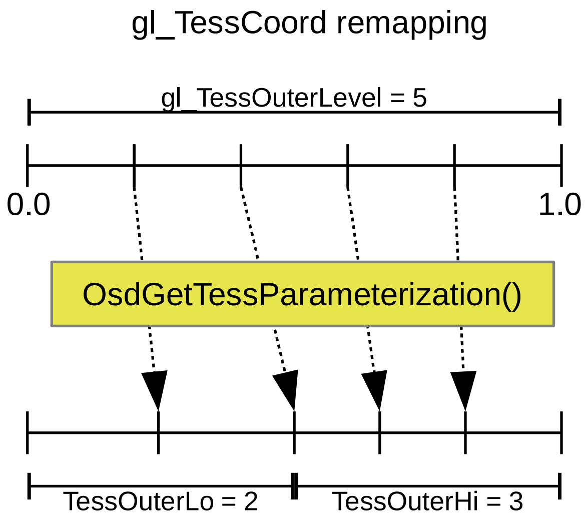


Figure 34: Osd shader patch parameter remapping

```
vec2 OsdGetTessParameterization(vec2 uv, vec4 tessOuterLo, vec4 tessOuterHi)
```

These tessellation levels can be computed by OsdGetTessLevelsAdaptiveLimitPoints() in the tessellation control shader. Note that this function requires all 16 bezier control points, you need to call barrier() to ensure the conversion is done for all invocations. See osd/glslPatchBSpline.glsl for more details.

```
void OsdGetTessLevelsAdaptiveLimitPoints(OsdPerPatchVertexBezier cpBezier[16],
                                         ivec3 patchParam,
                                         out vec4 tessLevelOuter, out vec2 tessLevelInner,
                                         out vec4 tessOuterLo, out vec4 tessOuterHi)
```

8 References

For more references, please visit http://graphics.pixar.com/opensubdiv/

RenderMan RIS

by Christophe Hery and Ryusuke Villemin

Figure 1: Photorealistic RenderMan 20 ©Disney/Pixar 2015

1 Introduction

For *Finding Dory*, the rendering pipeline at Pixar was completely rewritten to use RenderMan's new rendering engine RIS (Fig 1), in place of the old REYES renderer. Although ray-tracing and physically based shading were already used at Pixar, beginning with *Monsters University* and *The Blue Umbrella*, rendering was still done using the classical hybrid renderer (REYES+raytracing) with RSL coshaders[1]. The system was complex to maintain and couldn't achieve optimal performance because of this hybrid nature that needed to accommodate both rendering algorithms. With RIS, the switch to raytracing is now complete, and we have a modern, performant renderer, running fast C++ shaders, and can do efficient progressive rendering. More than ever, the setup complexity is handled by the computer, allowing the lighting artist to concentrate on the artistic side.

[1]A coshader is a specialized object construct in Pixar's RenderMan: for more information, refer to [The RenderMan Team 2009]

2 REYES/raytracing

Although the first use of raytracing for Pixar's productions goes back to *A bug's Life*, raytracing only became a main part of rendering (Fig 2) at Pixar since *Cars*, as outlined in [Christensen et al. 2006]. Lately it took a bigger step with the introduction of physically based shaders by [Hery and Villemin 2013] for *Monsters University* and *The Blue Umbrella*. However RenderMan remained a REYES renderer with some raytracing on top of it. Shading was primarily done using RSL, via a few extensions from its original inception in [Cook 1984]. As an interpreted language, RSL amortized its running cost by batching multiple points of a grid in parallel, and shading was completely detached from the visibility computation process. The advantage was that, we could have a different metric for the shading rate, and the visibility rate, and process rendering in independent small pieces, which was crucial decades ago when memory was a very limited resource. This shading method combined with subdivision surfaces from [DeRose et al. 1998] is the foundation of the REYES (Render Everything You Ever Saw) algorithm.

Figure 2: "A bug's life" 1998, "Cars" 2006, "Monsters University" 2012 ©Disney/Pixar 2015

Unfortunately now with raytracing, we have to shade where we hit, there is a very tight coupling between visibility and shading computation. Running RSL shaders for every ray hit (although some coherency can be achieved by spatially batching rays) proved to be too expensive. To solve this issue, the radiosity cache described in [Christensen et al. 2012] was introduced (Fig 3). The idea was to shade a whole grid of points at the first ray hit, and then reuse those results for subsequent hits. We basically had the grid shading back in raytracing, making the use of RSL possible again! The biggest caveat is that contrary to the primary hits all originating from the same direction, the render camera, indirect rays can hit the grid from any direction. Care is needed when storing data into the radiosity cache, basically only view-independent computations are valid, but they represent only a fraction of all possible computations. For many years, we lived with that limitation, by only using diffuse effects, and by keeping the use of specular effects in indirect hits only when absolutely necessary.

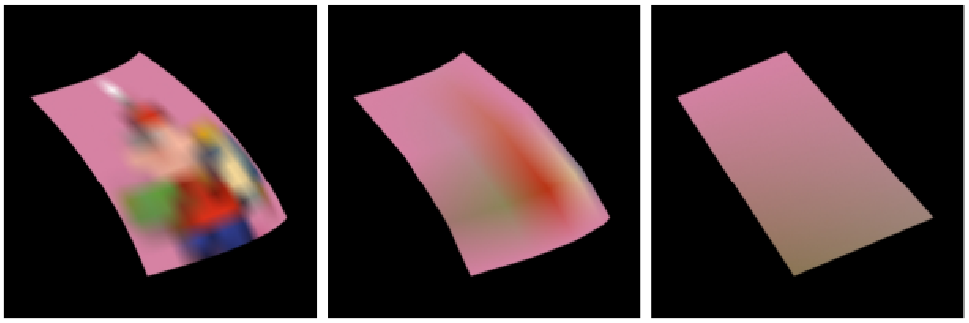
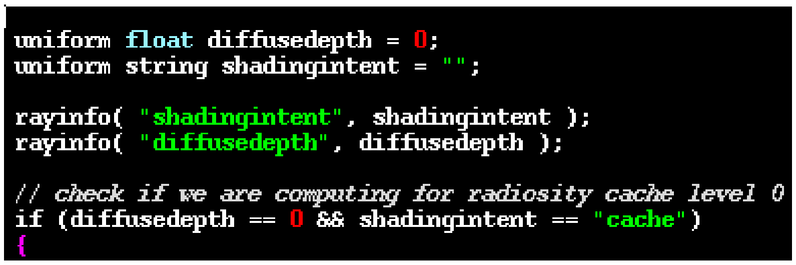


Figure 3: Multi level radiosity cache

The consequence of that hybrid system and its caching approach was an increased complexity in the shaders, which needed to run multiple tests and branching at every invocation, (Are we a primary hit? A secondary? Can we cache the current computation? See code example in Fig 4). The pain also extended to the users who had to decide when to cache, what to cache, at what resolution, making debugging harder when problems arose. And since there was no centralized management of all that intricacy, it was up to every shader to do the right thing, one bad shader in the scene potentially sending the whole render time to the roof, or worse producing incoherent results.



```
uniform float diffusedepth = 0;
uniform string shadingintent = "";

rayinfo( "shadingintent", shadingintent );
rayinfo( "diffusedepth", diffusedepth );

// check if we are computing for radiosity cache level 0
if (diffusedepth == 0 && shadingintent == "cache")
{
```

Figure 4: shader branching example

Because of the shading cost at every shader invocation, it was also very difficult to do photon and caustic tracing. They both require a full shader evaluation at every hit, which happens to be impractical in RSL. The compromise RenderMan took was a set of predefined models for shading photons (see [The RenderMan Team 2013]), namely refractive, chrome, matte and transparent, with very basic controls such as albedo and ior. This solved the speed, but basically required having two materials, the main one and the specialized simplified one for photons. The latter was not rich enough to match the complex shaders built in production, and, on top of that, required a manual synchronization with the main material.

3 Background

After a four year long transition period, the shift to raytracing is complete with the advent of a new renderer inside RenderMan, RIS. RIS is a modern, fully raytraced renderer. It runs fast C++ and OSL compiled shaders at every hit point, the grids are gone. Since there is no radiosity cache involved, we can implement standard raytracing techniques such as russian roulette, bidirectional tracing, etc. One of the main differences in RIS compared to REYES is that it requires a global integrator.

The main focus is the same as before, solve the rendering equation[2]:

$$L(x,\omega_o) = \int_\Omega f(x,\omega_i,\omega_o)L(x,\omega_i)cos(\theta)d\omega + L_e(x,\omega_o)$$

In this equation, there are two components: the Bxdfs f and the lights L, that are linked by the central conductor, the global integrator. A third implicit component is the visibility, and is now handled entirely by raytracing.

Typically, instead of trying to solve this equation in the general case, each surface shader was in charge of doing its own local integration. Now the global integrator will take care of this task, and thanks to its global view of the light paths, it can achieve more complex integration techniques, than each surface could do locally.

Previously each part of this integral was handled by a specific shader class, a coshader.

$$\begin{aligned}
L(x,\omega_o) &= \int_\Omega \left(f_{diffuse}(x,\omega_i,\omega_o) + f_{specular}(x,\omega_i,\omega_o)\right)\left(L_{direct}(x,\omega_i) + L_{indirect}(x,\omega_i)\right) cos(\theta)d\omega + L_e(x,\omega_o) \\
L(x,\omega_o) &= \int_\Omega f(x,\omega_i,\omega_o)L_{direct}(x,\omega_i)cos(\theta)d\omega + L_e(x,\omega_o) \\
&+ \int_\Omega f_{diffuse}(x,\omega_i,\omega_o)L_{indirect}(x,\omega_i)cos(\theta)d\omega \\
&+ \int_\Omega f_{specular}(x,\omega_i,\omega_o)L_{indirect}(x,\omega_i)cos(\theta)d\omega
\end{aligned}$$

that is :

$$L(x,\omega_o) = L_{direct} + L_{emission} + L_{indirectDiffuse} + L_{indirectSpecular}$$

$L(x,\omega_o)$	radiance from x in the ω_o direction $[Wm^{-2}sr^{-1}]$
$L(x,\omega_i)$	radiance to x in the ω_i direction $[Wm^{-2}sr^{-1}]$
$L_e(x,\omega_o)$	radiance emitted from x in the ω_o direction $[Wm^{-2}sr^{-1}]$
$f_s(x,\omega_i,\omega_o)$	surface BRDF at x from direction ω_i to direction ω_o $[sr^{-1}]$
θ	angle between the surface normal N and ω_i $[r]$

Table 11: Notations

L_{direct} was resolved by the directLighting integrator (lightPath = E{D,S}L, using [Heckbert 1990]'s path notation[3]). $L_{indirectDiffuse}$ was resolved by the indirectDiffuse integrator (lightPath = ED{D,S}*L), $L_{indirectSpecular}$ by the reflection integrator (lightPath = ES{D,S}*L). Splitting the computation, each having a specialized integrator was the key to get acceptable performance, but also a curse because each required a special setup, and it was very difficult to get enough flexibility to use all of them in combination.

[2]In its simplified form here.

[3]Eye(E), Specular(S), Diffuse(D), Light(L)

In RIS, we don't have this splitting required anymore, but computing the rendering equation in the general form is quite a challenge. One of the main problems is that if we do not decompose the equation like before, L ends up in both sides of the equation. To make this easier to solve, let's rewrite the equation as a Neumann expansion series using the integral operator notation T:

$$< Tg > (x, \omega_o) = \int_\Omega f(x, \omega_i, \omega_o) g(x, \omega_i) cos(\theta) d\omega$$

then:

$$L = TL + L_e$$

If we recursively expand it:

$$L = T\left(L_e + T(L_e + T(L_e + T...TL_e...))\right)$$

or

$$L = \sum_{n=0}^{\infty} T^n L_e$$

The global integrator can use the appropriate notation to solve the equation. In RIS, the integrator is a completely customizable plugin, so almost any integrator could be implemented, although RenderMan provides by default a performant pathtracer, and a more advanced UPS [Hachisuka et al. 2012] /VCM [Georgiev et al. 2012] one.

4 RIS

4.1 Overview

The rendering computation can be decomposed into four main parts (Fig 5):

- RixPattern: The patterns don't directly contribute to the lighting integration but they are responsible for feeding spatially varying signals (like textures) to the Bxdfs. They are written in C++ or OSL (more info at [Gritz 2009]).
- RixLightingServices: The LightingServices will abstract all the lights in the scene and provide light samples to the global Integrator. It is currently handled by RenderMan and is not customizable except in specific places using LightFilters.
- RixBxdf: The Bxdfs shaders are responsible for providing all the surface samples with respect to their sampling strategies. The same way the lighting services have to provide a number of samples independent of the number of lights in the scene, each Bxdf needs to be capable of providing one sample per call independently of the number of lobes it contains.
- RixIntegrator: The integrator handles the light path construction from the camera to the light sources, querying and evaluating the lighting services and the Bxdfs.

All the above classes are customizable as inherited classes from the base classes by overloading specific virtual functions.

One remarkable point is that although RenderMan does not shade grids anymore, the tracing engine will bundle rays, and also hit points, so that multiple rays hitting the same object will call the material in batches making vectorization particularly easy (see Fig 6). The terminology is kept the same as in REYES, *varying* for variables dependent on the hit point, *uniform* for independent variables. At Pixar, we take advantage of that parallelization by using ISPC from [Pharr and Mark 2012] when writing our plugins.

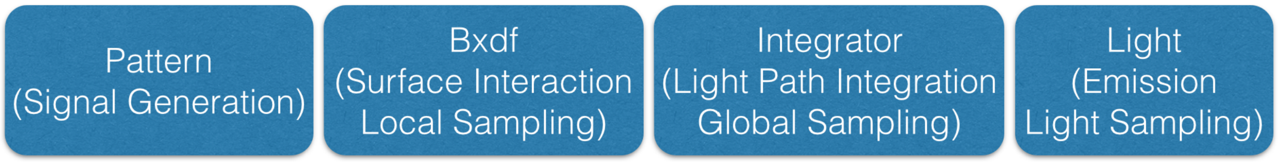


Figure 5: Separation between patterns, Bxdfs and integrator

The shading context, *RixShadingContext* (defined in Listing 1) is the element that is passed from the integrator to the lighting services, Bxdfs, light filters and patterns, and contains all the information about the current hits (P, Nn, curvature, ...).

Listing 1: RixShadingContext

```
class RixShadingContext : public RixContext
{
public:

    enum BuiltinVar
    {
        k_P=0,
        k_PRadius,
        k_Po,       // undisplaced P
        k_Nn,       // normalized shading normal
        k_Ngn,      // normalized geometric normal
        k_Non,      // undisplaced N
        k_Tn,       // normalized shading tangent
        k_Vn,       // normalized view vector, points away from shading points
        k_VLen,     // length of V: from the P to previous P
        k_curvature, // local surface curvature
        ...
    }

    virtual void GetBuiltinVar(BuiltinVar, RtInt const**var) const = 0;
    virtual void GetBuiltinVar(BuiltinVar, RtFloat const**var) const = 0;
    virtual void GetBuiltinVar(BuiltinVar, RtFloat3 const**var) const = 0;

    ...
};
```

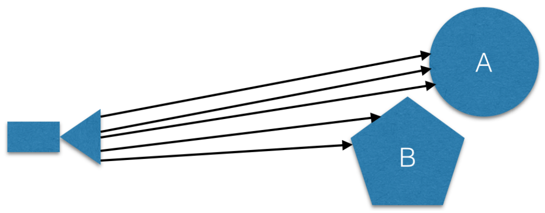


Figure 6: A five ray batch generating two shader calls on two shading contexts A and B

4.2 RixPattern

A pattern shader's main function is *ComputeOutputParams* and returns one or multiple results, which can be in turn varying or uniform. In the Listing 2 example, ColorMix only returns a single output which is a varying array of colors. Patterns can be chained and connected through a DAG[4]. The evaluation of incoming connections is done through *EvalParam*. This function will automatically trigger the evaluation of any upstream pattern shaders, or just return a constant value if directly specified in the RIB file (and a default value if not). The output array is allocated using RenderMan's stack allocator which will be optimized to not do any heap allocation during rendering, and also have an automatic deallocation at the end of the shading context lifetime. This also serves as a cache for the lifetime of the shading context, so if multiple patterns are connected to the same one, the evaluation will happen only once.

Listing 2: Color Mix Pattern

```
RtInt ComputeOutputParams( RixShadingContext const *shadingCtx,
                           RtInt *nOutputs, OutputSpec **outputs,
                           RtConstPointer instanceData,
                           RixSCParamInfo const *pInfo )
{
    RtInt numPts = shadingCtx->numPts;
    const RtColorRGB *ColorA = 0x0;
    shadingCtx->EvalParam( k_ColorA, -1, &ColorA, &_ColorA, true );

    const RtColorRGB *ColorB = 0x0;
    shadingCtx->EvalParam( k_ColorB, -1, &ColorB, &_ColorB, true );

    const RtFloat *mixer = 0x0;
    shadingCtx->EvalParam( k_mixer, -1, &mixer, &_mixer, true );

    // allocate and bind our outputs
    RixShadingContext::Allocator pool(shadingCtx);
    OutputSpec *o = pool.AllocForPattern<OutputSpec>(1);
    *outputs  = o;
    *noutputs = 1;

    RtColorRGB *ResultC = pool.AllocForPattern<RtColorRGB>(numPts);

    o[0].paramId = k_ResultC;
    o[0].detail  = k_RixSCVarying;
    o[0].value   = (RtPointer)ResultC;

    for( RtInt pointId = 0; pointId < numPts; ++pointId )
    {
      RtFloat bAmt = mixer[pointId];
      RtFloat aAmt = 1.f - bAmt;
      ResultC[pointId] = aAmt * ColorA[pointId] + bAmt * ColorB[pointId];
    }

    return 0;
}
```

[4]Directed Acyclic Graph

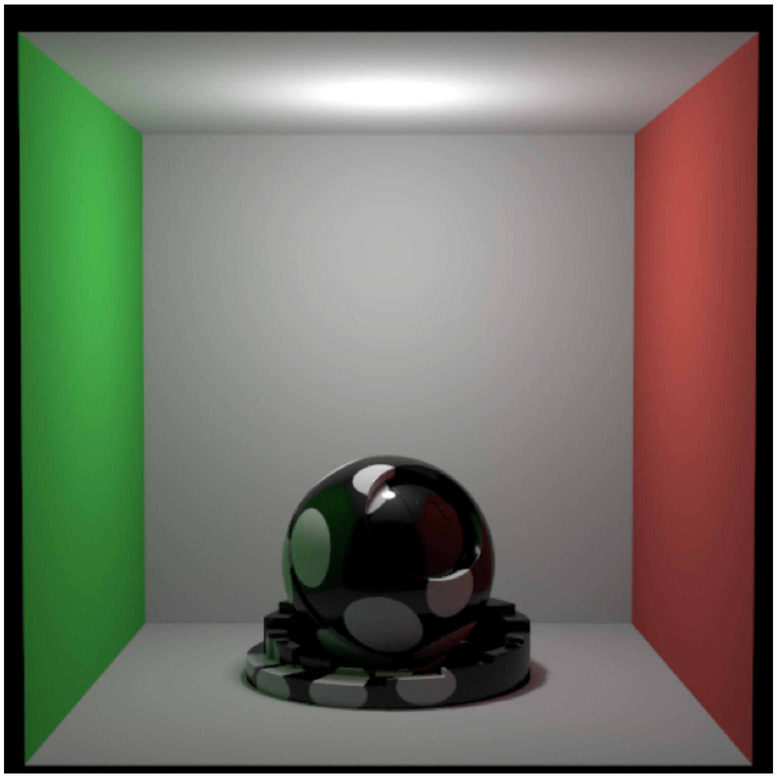


Figure 7: Procedural pattern

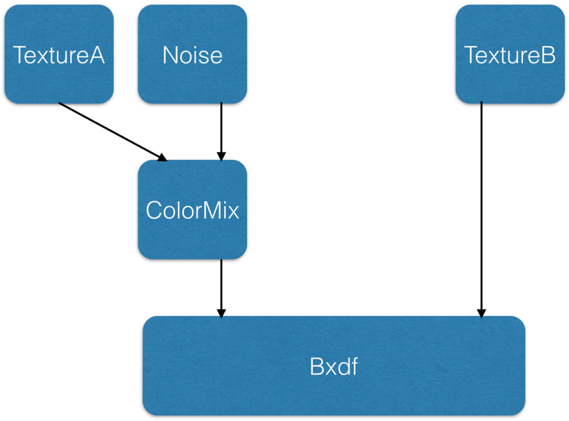


Figure 8: Procedural pattern network connected to a Bxdf

4.3 RixBxdf

There is one and only one Bxdf attached to each object in the scene. The Bxdf is the connection between the object and the integrator and it needs to provide two functions to the integrator. For a given input and output direction, provide the value and corresponding pdfs, but also needs to be able to generate a new direction based on the current incoming direction. The corresponding two functions are *GenerateSample* and *EvaluateSample*. Any required data from the shading context are queried via *GetBuiltinVar*. The inputs are given by a pattern or a graph of patterns shaders through *EvalParam*.

Listing 3: Lambertian Bxdf Sample

```
void GenerateSample( RixBXTransportTrait transportTrait,
                     RixBXLobeTraits const *lobesWanted,
                     RixRNG *rng,
                     RixBXLobeSampled *lobeSampled,
                     RtVector *Ln,
                     RixBXLobeWeights &W,
                     RtFloat *fPdf, RtFloat *rPdf )
{
    //Get shading context variables
    RtInt numPts = shadingCtx->numPts;

    const RtNormal* Nn = 0x0;
    shadingCtx->GetBuiltinVar( RixShadingContext::k_Nn, &Nn );
    const RtVector* Vn = 0x0;
    shadingCtx->GetBuiltinVar( RixShadingContext::k_Vn, &Vn );
    const RtVector* Tn = 0x0;
    shadingCtx->GetBuiltinVar( RixShadingContext::k_Tn, &Tn );

    // allocate and init the result array
    RtColorRGB *diffuse = W.AddActiveLobe( _diffuseLobe );

    //Draw random numbers
    RtFloatTwo xi[nPts];
    rng->DrawSamples2D( nPts, xi );

    for( RtInt pointId = 0; pointId < numPts; ++pointId )
    {
      if( lobeWanted[pointId] & _lambertLobeTraits )
      {
        RtFloat VdotN = Dot( Vn[pointId], Nn[pointId] );
        if( VdotN > 0.f )
        {
          // cosine based distribution
          RtFloat LdotN;
          RixCosDirectionalDistribution( xi[pointId], Nn[pointId], On[pointId], LdotN );

          fPdf[pointId] = LdotN * F_INVPI;
          rPdf[pointId] = VdotN * F_INVPI;
          diffuse[pointId] = LdotN * F_INVPI * _albedo[pointId];
          lobeSampled[pointId] = _lambertLobe;
        }
        else
        {
          lobeSampled[pointId].SetValid( false );
        }
      }
      else
      {
        lobeSampled[pointId].SetValid( false );
      }
    }
}
```

Listing 4: Lambertian Bxdf Evaluate

```
void EvaluateSample( RixBXTransportTrait transportTrait,
                     RixBXLobeTraits const *lobesWanted,
                     RixBXLobeTraits *lobesEvaluated,
                     const RtVector *Ln,
                     RixBXLobeWeights &W,
                     RtFloat *fPdf, RtFloat *rPdf )
{
    // get shading context variables
    RtInt numPts = shadingCtx->numPts;

    const RtNormal* Nn = 0x0;
    shadingCtx->GetBuiltinVar( RixShadingContext::k_Nn, &Nn );
    const RtVector* Vn = 0x0;
    shadingCtx->GetBuiltinVar( RixShadingContext::k_Vn, &Vn );

    // allocate and init the result array
    RtColorRGB *diffuse = W.AddActiveLobe( _diffuseLobe );

    for( RtInt pointId = 0; pointId < numPts; ++pointId )
    {
      if( lobeWanted[pointId] & _lambertLobeTraits )
      {
        RtFloat VdotN = Dot( Vn[pointId], Nn[pointId] );
        RtFloat LdotN = Dot( Ln[pointId], Nn[pointId] );
        if( VdotN > 0.f && LdotN > 0.f )
        {
          fPdf[pointId] = LdotN * F_INVPI;
          rPdf[pointId] = VdotN * F_INVPI;
          diffuse[pointId] = LdotN * F_INVPI * _albedo[pointId];
          lobesEvaluated[pointId] = _lambertLobeTraits;
        }
        else
        {
          lobesEvaluated[pointId].SetNone();
        }
      }
      else
      {
        lobesEvaluated[pointId].SetNone();
      }
    }
}
```

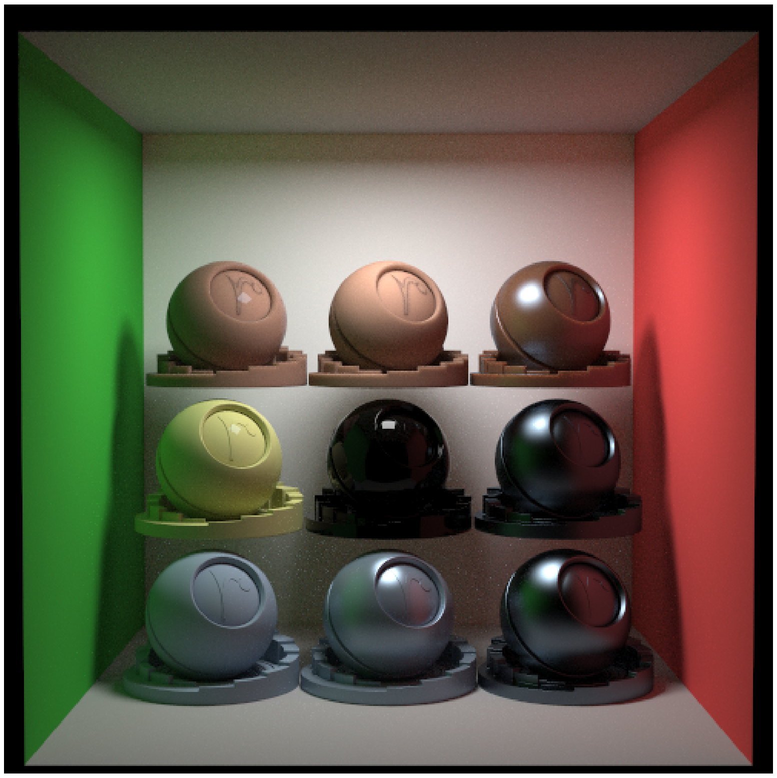
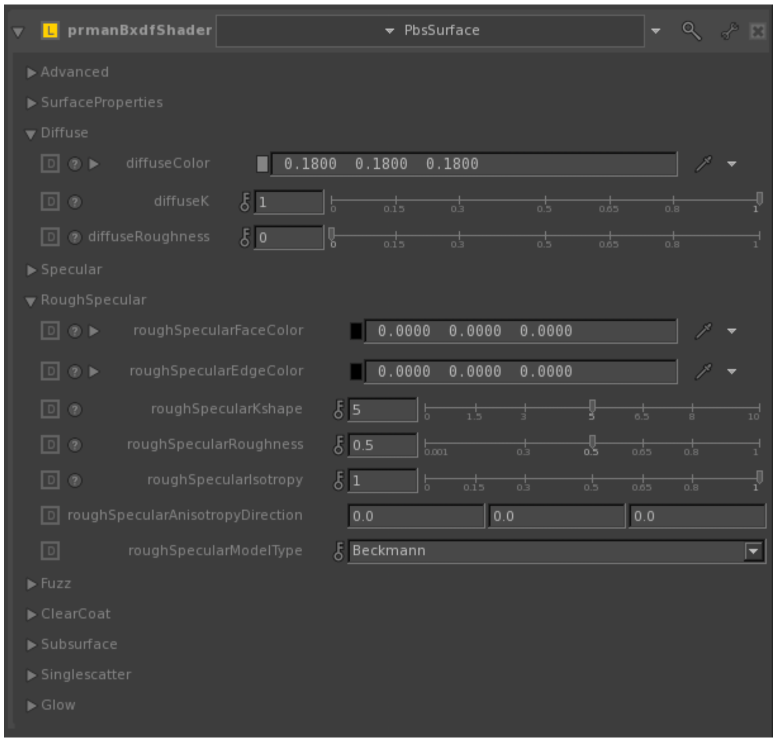


Figure 9: Various Bxdfs

Although the integrator will see only one Bxdf, it doesn't mean that this Bxdf needs to be simple. In production, a Bxdf shader tends to be an aggregate of multiple "lobes", the standard one illustrated in Fig 10 used at the studio contains:

- 1 diffuse
- 3 speculars
- 1 fuzz
- 1 glow
- 1 subsurface
- 1 singlescatter

This uber Bxdf contains additional logic to efficiently sample between all the lobes. We achieve that by using one sample MIS [Veach and Guibas 1995] between them. For instance, when sampling the specular lobe of a Bxdf, we also evaluate the diffuse part of it, which results in the variance reduction shown in Fig 11.


prmanBxdfShader PbsSurface

- Advanced
- SurfaceProperties
- Diffuse
  - diffuseColor 0.1800 0.1800 0.1800
  - diffuseK 1
  - diffuseRoughness 0
- Specular
- RoughSpecular
  - roughSpecularFaceColor 0.0000 0.0000 0.0000
  - roughSpecularEdgeColor 0.0000 0.0000 0.0000
  - roughSpecularKshape 5
  - roughSpecularRoughness 0.5
  - roughSpecularIsotropy 1
  - roughSpecularAnisotropyDirection 0.0 0.0 0.0
  - roughSpecularModelType Beckmann
- Fuzz
- ClearCoat
- Subsurface
- Singlescatter
- Glow

Figure 10: PbsSurface

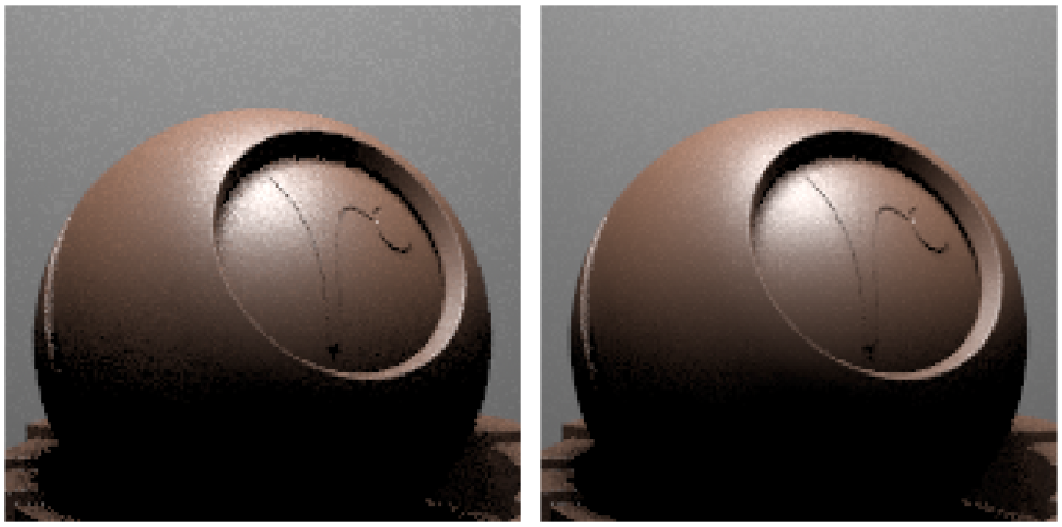

(a) no MIS (b) One sample MIS between lobes

Figure 11: Multi-lobe MIS

4.4 RixLightFilter

LightFilters are linked to individual lights, and because of their nature, they operate a little differently than Bxdfs and Patterns. Instead of working directly on a RixShadingContext, they will operate on a specialized RixLightFilterContext that corresponds to a compacted version of the shading context, containing only the points that are querying the same light. The example we give here is a simple spatial rod filter that will modify and tint all the diffuse and specular contributions of the light it is attached to.

Listing 5: Light Filter Rod

```
void Filter( RixLightFilterContext const *lfCtx,
             RtConstPointer instanceData,
             RtInt const numSamples,
             RtInt const *shadingCtxIndex,
             RtVector3 const *toLight,
             RtFloat const *dist,
             RtFloat const *lightPdfIllum,
             RixBXLobeWeights *contribution )
{
    if ( _density != 0.f )
    {
      RtPoint* xformedP = TransformPIntoLocalSpace( lfCtx, numSamples, shadingCtxIndex );

      RtInt nbDiffs = contribution->GetNumDiffuseLobes();
      RtInt nbSpecs = contribution->GetNumSpecularLobes();

      for( RtInt sampleId = 0; sampleId < numSamples; ++sampleId )
      {
        const RtPoint& point = xformedP[sampleId];
        RtFloat contrib = 0.f;
        RtColorRGB tint(1.f);

        _rod.run( point, contrib, tint );

        // apply the contribution
        RtColorRGB diffuse ( 1.f );
        RtColorRGB specular( 1.f );

        if( contrib > 0.f )
        {
          diffuse  *= RixMix( 1.f, _diffMult, contrib );
          specular *= RixMix( 1.f, _specMult, contrib );
        }

        diffuse  *= RixMix( RixConstants::k_OneRGB, tint, _density );
        specular *= RixMix( RixConstants::k_OneRGB, tint, _density );
      }
    }
}
```

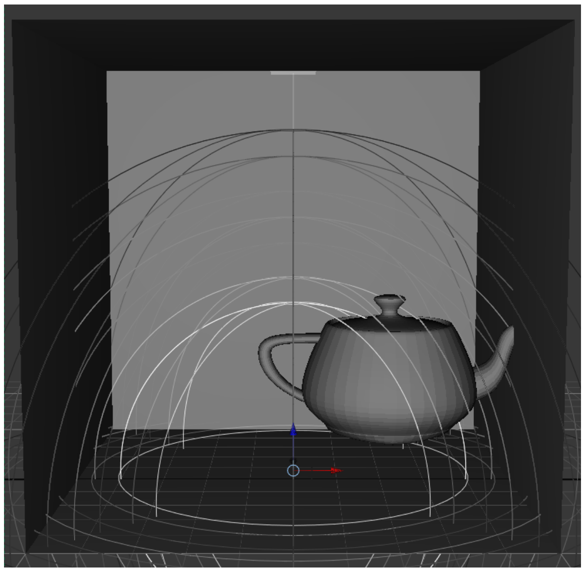
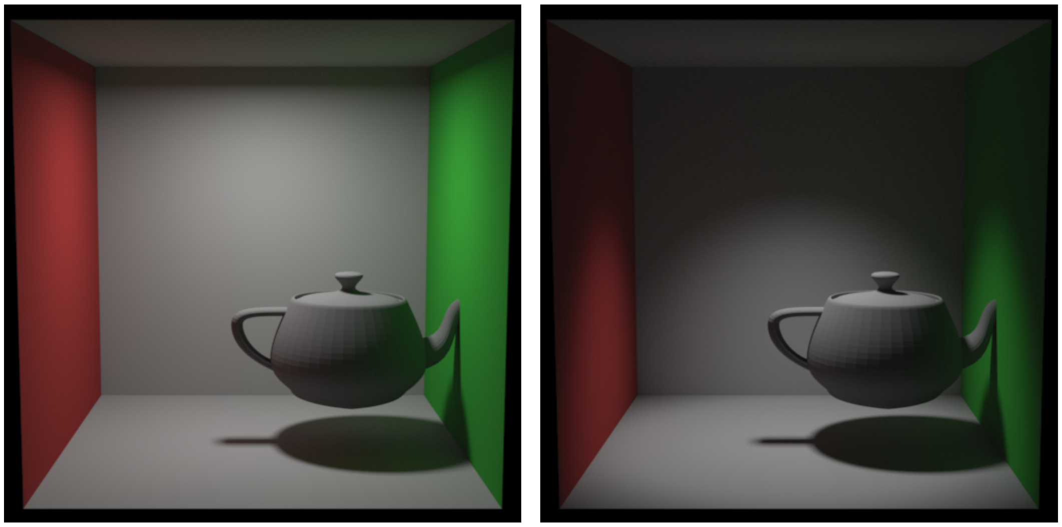


Figure 12: Rod Preview in Katana



(a) no Rod

(b) Rod Filter Attenuation

Figure 13: Rod Light Filter

4.5 RixIntegrator

The integrator is the central part of the rendering. Its job begins at the primary shading contexts.

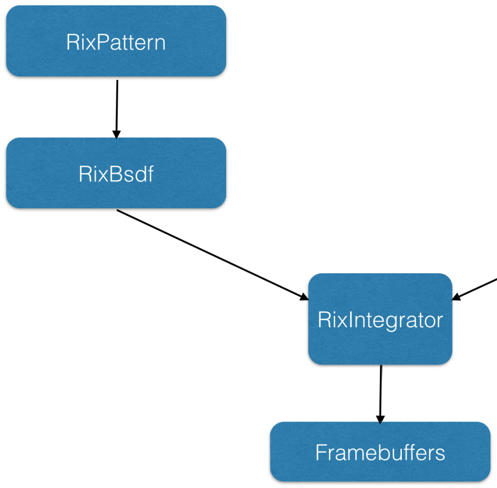
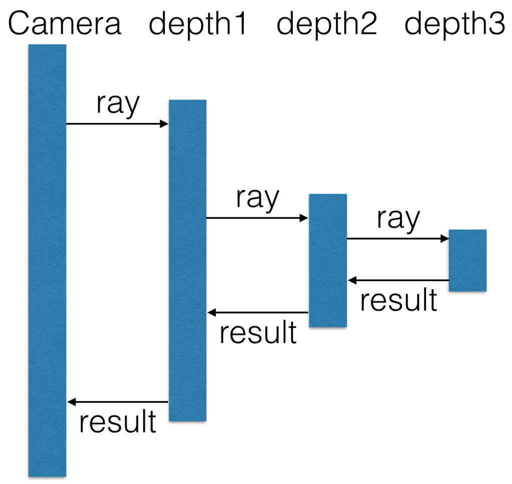
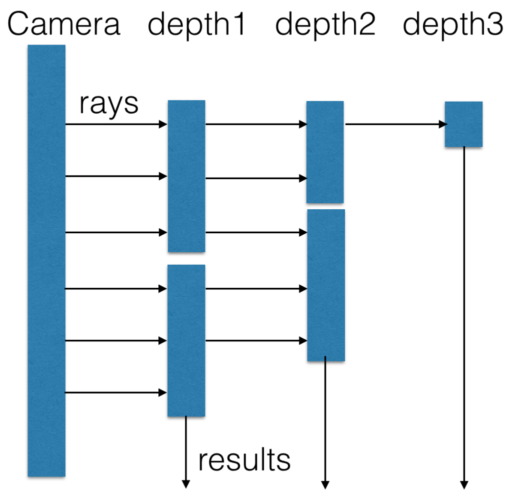


Figure 14: RixIntegrator

The simplest integrator could be a recursive raytracer that traces one ray at a time, every hit calling recursively the next trace call, but this wouldn't be very efficient. Instead RIS offers the possibility of tracing multiple rays at once, as waves of rays, leading to faster tracing, as well as less shader calls. The downside is that processing batches of points usually requires more memory than single point, but this increased memory can be reduced by writing an iterative raytracer that will flush out local memory every bounces. To illustrate, Fig 15a represents the diagram of operations if we recursively want to compute the 3rd bounce recusively (notation: $L = T(T(TLe)))$). On the other hand, we can do it iteratively, in the manner of Fig 15b. In the first case, we need to keep around all the bounces before we can multiply by L_e, in the latter when we arrive at L_e we already have the correct thruput T^3.



(a) Recursive Raytracing



(b) Iterative Raytracing

Figure 15: Iterative/Recursive raytracing

RenderMan passes information needed through the bounces using the *RtRayGeometry* class. If an integrator needs additional data, it can keep a custom payload per ray (see Fig 16), contaning for example the current thruput of the ray, that can be used to do russian roulette or termination threshold.

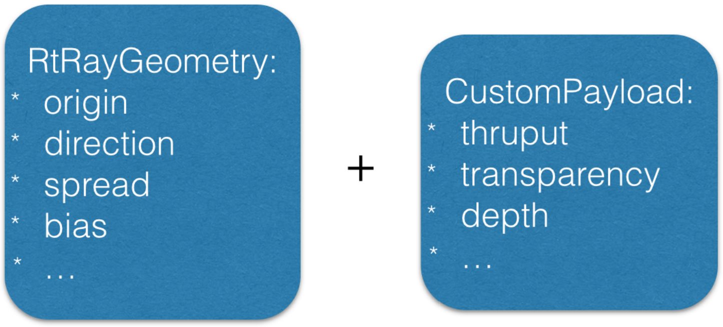
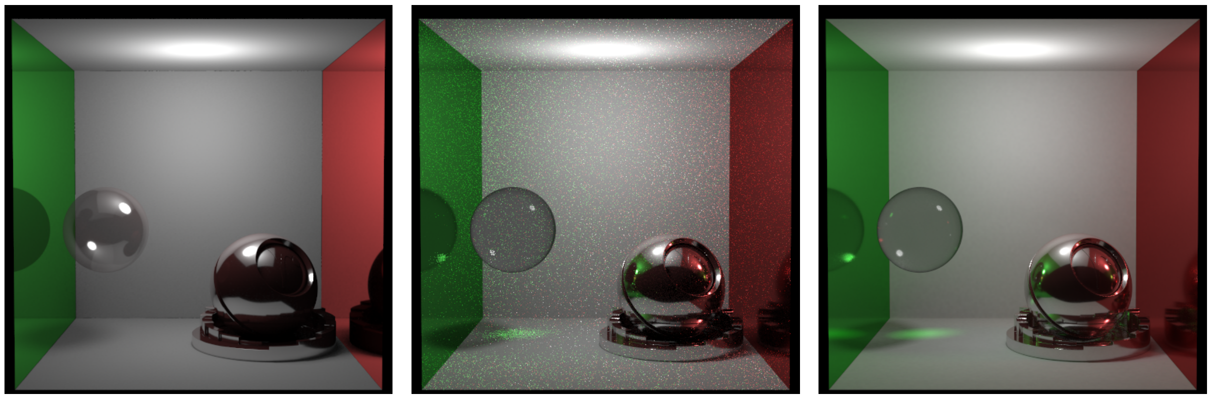


Figure 16: Custom Ray Payload

Depending on the integration method, some light paths are not correctly sampled and can create high variance (SDS or even standard caustic paths). There are two ways of solving this, use a more complex integrator, or just detect and ignore those contributions. Both methods are available though the default integrators (Fig 17b and 17c) and produce better results than the old REYES render. Notice the missing reflection, refractions because the use of the radiosity cache (Fig 17a), and the nice reflected caustics in the RIS bidirectional render.

If we decide to remove the high variance contributions during render, we usually apply a simple clamp to reduce the fireflies. We also can leave them for the post process denoiser to remove. In any case, we try to introduce as little specialized code as possible in the integrator, to leave the door open to new integration techniques that may not be compatible with those special cases.



(a) Reyes, radiosity cache raytracing (b) unidirectional path tracing (c) bidirectional path tracing

Figure 17: REYES vs RIS raytracing

4.6 A quick note about bi-directionality

As we have seen, the visual effects and animation rendering practionners have, for many years, employed various tricks to produce images. With the advent of physically based illumination, a lot more structure was introduced in how Bxdfs and lights are defined. Even then, we are only now entering, in production, the era of bi-directional integration. This comes with yet additional constraints.

Let's take an example. In [Hery and Villemin 2013], at paragraph 3, we described the main specular model [5] that we designed for *Monsters University* and pretty much used universally at Pixar until now, even for the upcoming movie *The Good Dinosaur.* For reference, here is its equation:

$$f_\mu(\mathbf{l}, \mathbf{v}) = \frac{F(\mathbf{l}, \mathbf{h})\, D_b(\mathbf{h})\, (\mathbf{n} \cdot \mathbf{h})}{4\, (\mathbf{n} \cdot \mathbf{l})\, (\mathbf{v} \cdot \mathbf{h})}, \text{with } D_b \text{ being the Beckmann distribution.}$$

Our main concern at the time was about limiting computation (because of relatively slow RSL execution), easy sampling ensuring convergence with MIS under direct lighting as well as tracing for specular reflections, and energy conservation. This last item was crutial to satisfy the intuition of the lighting and shading artists, who were used to ad-hoc enviromnent mapping as described in [Blinn and Newell 1976], where a white uniform latlong image was expected to produce a uniform response from the reflection environment look-up. Remember that the hybrid Reyes/ray-trace mode of RenderMan mainly corresponds to a forward distributed ray-tracing approach, where events are solved only from the camera onwards. As a consequence, we never cared about reciprocity.

So we gave up reciprocity alltogether in favor of absolute energy conservation. You can see that from the equation above, where $\mathbf{l}$ and $\mathbf{v}$ cannot be interchanged to produce the same result. Another thing we did, because of not directly sampling the half-way vector space, some sampled directions would end up under the horizon, where they would be wasted and potentially would result in a slight loss of energy. To overcome that, we did not normalize to the full number of samples in the integrator, only to the valid ones. Note that this made our model even less reciprocal (at the same time it enforced energy conservation).

With RIS and in particular bi-directional integration, special care needs to be applied to non reciprocal models as in [Veach 1996], or better, we should strive to employ only reciprocal models. At this stage, even with the recent strategies in visible normal sampling laid out in [Heitz 2014], we cannot satisfy absolute energy conservation in known microfacet models, especially with large input roughnesses. [To be frank, now that artists are used to physically based rendering and do not refer anymore to their old habits, this is not even an issue.] Recently, [Jakob 2015] observed this phenomenon and proposed an approximate rectification factor. See comparison in Fig 18.

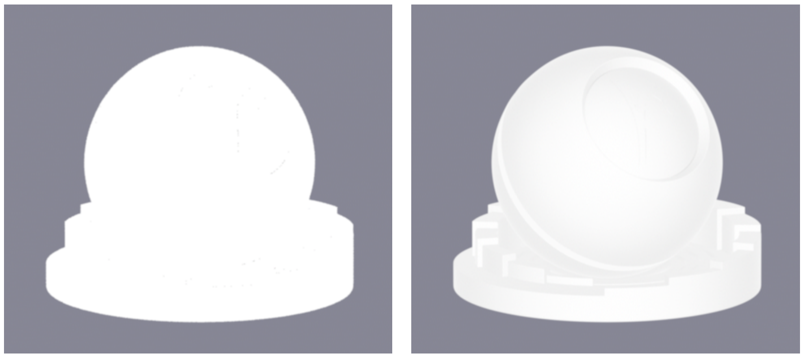
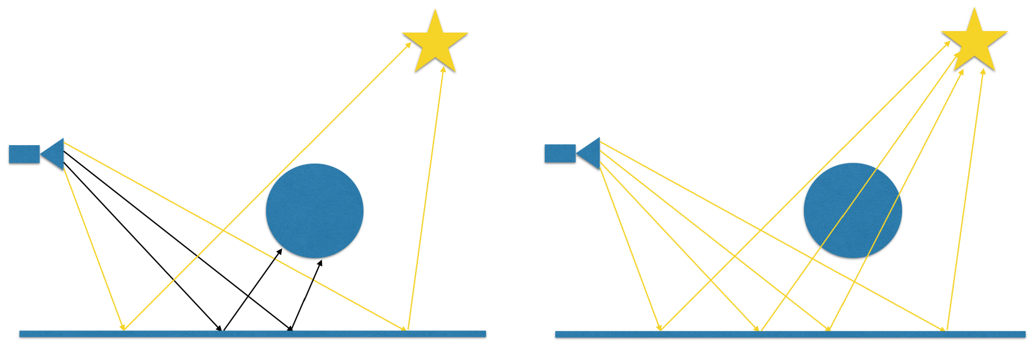

(a) Monsters University model (b) Physically based model

Figure 18: Furnace test with roughness 1. Note the slight loss of energy on the reciprocal model.

[5]Some people have named this simple model ”the LEAN mapping BRDF”.

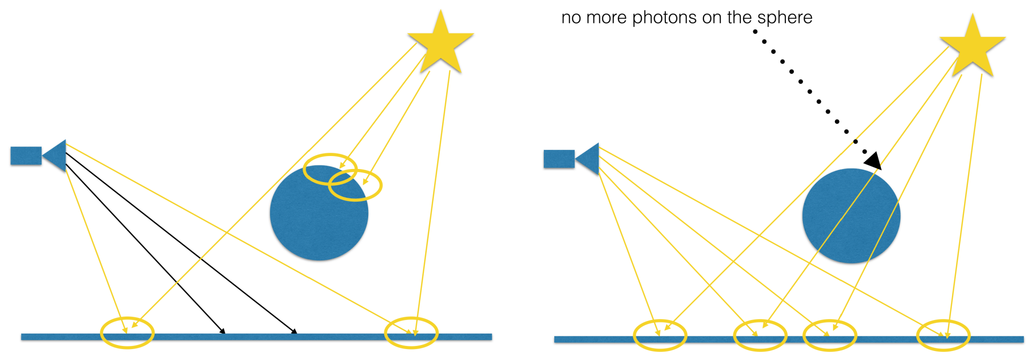
One other point that can be difficult to assimilate is the change in the concept of shadows. Previously shadows were seen as a multiplier on the lighting response. Especially for people coming from the REYES algorithm, where historically (see [Reeves et al. 1987]) shadows were a separately computed map, that was multiplied on top on the "non-shadowed" lighting result. This is not true anymore in bidirectional pathtracing or photon tracing, where shadows are just the natural result of the absence of light. Explicit shadow linking becomes impossible.



(a) Light rays are blocked by the sphere

(b) The sphere is invisible to the light rays

Figure 19: Unlinking shadows.



(a) Photons accumulate on the ground and the sphere and a shadow forms naturally under the sphere.

(b) Photons do not see the sphere, this removes the shadow, but at the same the illumination on the sphere!

Figure 20: Classical shadow linking failure in photon.

A large amount of the old tricks are not available in bidirectional anymore. Some of them are not needed because of the interactive rendering that allows the lighter set up lighting more easily than before. Nonetheless, we are currently looking into new forms of controls we can give the lighters, for instance through making use of LPEs [6].

[6]Light Path Expressions

5 Katana pipeline integration

At the same time rendering switched from REYES to RIS, the lighting tool shifted from our internal tool Lumos to Katana. The main benefits from that move are:

- shading is now also done in Katana, so artists can now shade in context. The shading network is completely accessible in Katana (Fig 21), and we don't have to jump to another software to shade and to light.
- lighting can now use Katana's interactive rendering (Fig 22). The lighters can change parameters on the lights, position and orientation, but also even shader parameters and camera. Once the live render is on, every edit happening on the scene will automatically show up in the render in realtime without having to pay for an expensive and manual restart.

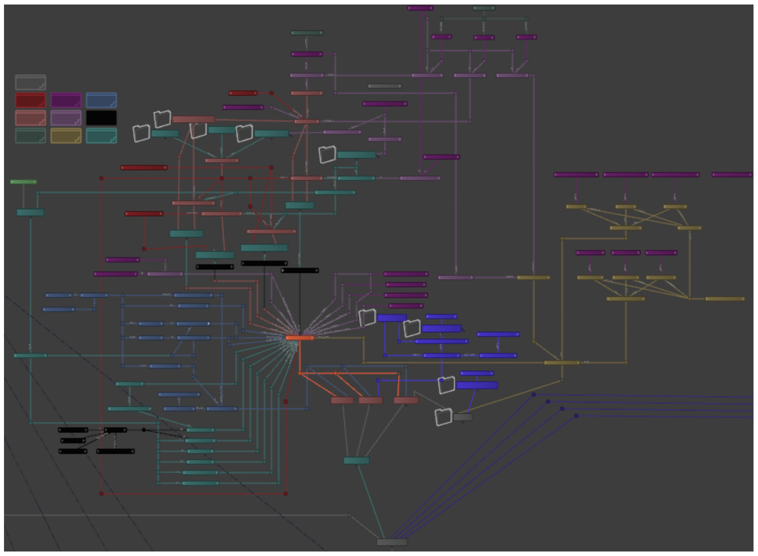


Figure 21: Complex shading network in Katana

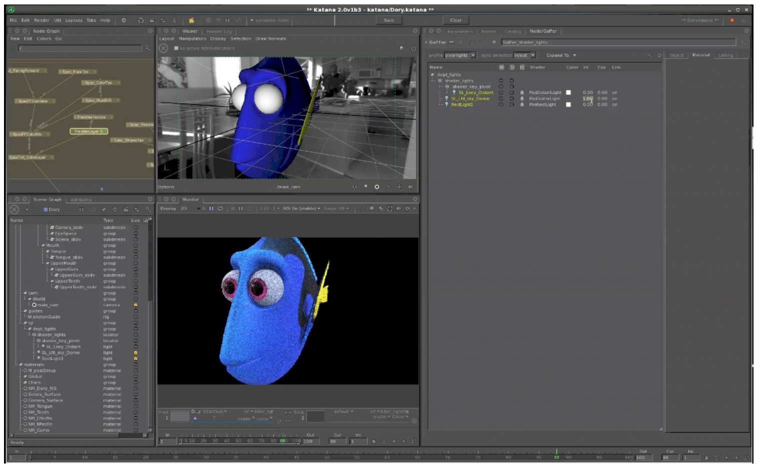


Figure 22: LiveRendering in Katana

Since RIS is using raytracing from the primary rays starting at the camera, and the render is progressive, it is now possible to have a different number of samples per pixel. This is called adaptive rendering. RenderMan internally keeps a buffer containing the current variance and error for each pixel, and uses that to stop or continue to shoot rays within that pixel. The user can control the quality by specifying a target error threshold or a target rendertime. In the Fig 23 example, we can see that most of the computation went in the splashes, that require more reflection and refraction events to converge, but the rest of the image (the still water) stopped much faster. At the end of the render, RenderMan reports the average and number of active pixels per iteration.Various image buffers can be incorporated in the final variance computation. For example, by using custom AOVs, we can force the renderer to shoot more rays on specific objects or regions.

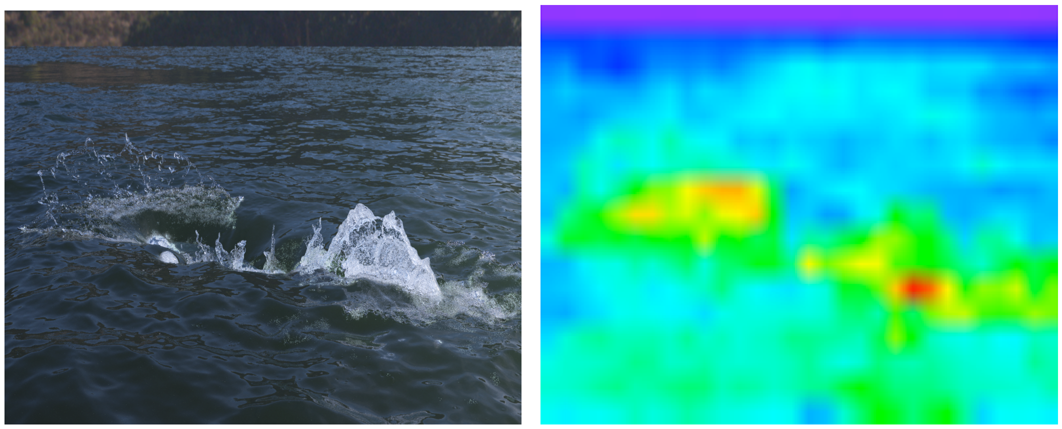

(a) Adaptive Rendering (b) Adaptive Heatmap

Figure 23: Adaptive Rendering

Another benefit of the progressive nature of the rendering is checkpointing. Controlled by either a number of iterations or just render time intervals, we can tell RenderMan to write intermediate images on disk, so we can now have very quickly an estimate of what the final images are going to look like (Fig 24). This means that we can detect if anything is wrong in the image ("I forgot a light!") without waiting for the full rendertime, and we can look at a full sequence even if all the images have not fully finished yet.

Checkpointing can be restarted. So, if an image ends up too noisy, we can resume with additional iterations and let the render converge from there.

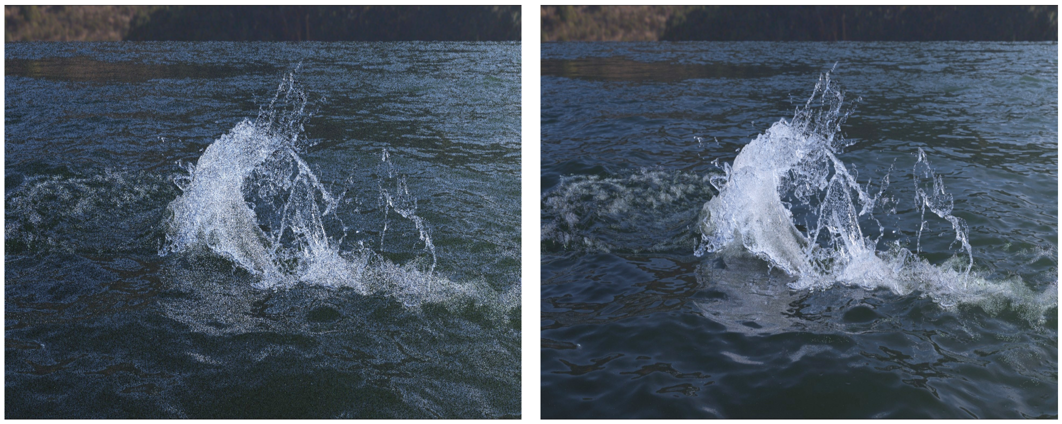

(a) After 1 hour (b) After 10 hours

Figure 24: Render Checkpoints

6 Conclusion

Moving to RIS overall improved the lighting efficiency by enabling faster iterations, which translated in more creative exploration and refinements. Although we have a completely new renderer, the RenderMan spirit is still alive, by providing a complete, open API, where every user can customize the renderer as she wishes.

From a developer point of view, the advantages are multiple. Because there is no real limitation on the API, it is now much easier to implement and test the latest Siggraph papers on rendering. The move to C++ also makes porting to and from other renderers easier, even cross platform, for example in Optix and CUDA, a very C like language.

We want to thank the full RenderMan team, who we worked with more closely than ever to stress the new RIS API, and with which we are currently collaborating to ship the studio shaders externally. We also want to acknowledge here the work of all the shading, lighting and rendering TDs in the RUKUS project on *Finding Dory*, whose mission was to bring RIS and Katana into the studio pipeline.

References

[Blinn and Newell 1976] Blinn, J. F., and Newell, M. E. 1976. Texture and reflection in computer generated images. In *Proceedings of the 3rd Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '76, 266–266.

[Christensen et al. 2006] Christensen, P., Fong, J., Laur, D., and Batali, D. 2006. Ray tracing for the movie 'cars'. *Symposium on Interactive Ray Tracing 0*, 1–6.

[Christensen et al. 2012] Christensen, P. H., Harker, G., Shade, J., Schubert, B., and Batali, D. 2012. Multiresolution radiosity caching for global illumination in movies. In *ACM SIGGRAPH 2012 Talks*, ACM, New York, NY, USA, SIGGRAPH '12, 47:1–47:1.

[Cook 1984] Cook, R. L. 1984. Shade trees. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '84, 223–231.

[DeRose et al. 1998] DeRose, T., Kass, M., and Truong, T. 1998. Subdivision surfaces in character animation. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '98, 85–94.

[Georgiev et al. 2012] Georgiev, I., Křivánek, J., Davidovič, T., and Slusallek, P. 2012. Light transport simulation with vertex connection and merging. *ACM Trans. Graph. 31*, 6 (Nov.), 192:1–192:10.

[Gritz 2009] Gritz, L., 2009. Open shading language. `http://opensource.imageworks.com/?p=osl`.

[Hachisuka et al. 2012] Hachisuka, T., Pantaleoni, J., and Jensen, H. W. 2012. A path space extension for robust light transport simulation. *ACM Trans. Graph. 31*, 6 (Nov.), 191:1–191:10.

[Heckbert 1990] Heckbert, P. S. 1990. Adaptive radiosity textures for bidirectional ray tracing. *SIGGRAPH Comput. Graph. 24*, 4 (Sept.), 145–154.

[Heitz 2014] Heitz, E. 2014. Understanding the masking-shadowing function in microfacet-based brdfs. *Journal of Computer Graphics Techniques (JCGT) 3*, 2 (June), 32–91.

[Hery and Villemin 2013] Hery, C., and Villemin, R., 2013. Physically based lighting at pixar. `http://graphics.pixar.com/library/PhysicallyBasedLighting/index.html`.

[Jakob 2015] Jakob, W., 2015. Layerlab: A computational toolbox for layered materials. `http://blog.selfshadow.com/publications/s2015-shading-course/`.

[Pharr and Mark 2012] Pharr, M., and Mark, W. R., 2012. ispc: A SPMD compiler for high-performance cpu programming. `http://llvm.org/pubs/2012-05-13-InPar-ispc.html`.

[Reeves et al. 1987] Reeves, W. T., Salesin, D. H., and Cook, R. L. 1987. Rendering antialiased shadows with depth maps. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '87, 283–291.

[The RenderMan Team 2009] The RenderMan Team, 2009. Shader objects and co-shaders. `http://renderman.pixar.com/resources/current/rps/shaderObjects.html`.

[The RenderMan Team 2013] The RenderMan Team, 2013. New photon mapping features. `http://renderman.pixar.com/resources/current/rps/newPhotonMapping.html`.

[Veach and Guibas 1995] Veach, E., and Guibas, L. J. 1995. Optimally combining sampling techniques for monte carlo rendering. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, SIGGRAPH '95, 419–428.

[Veach 1996] Veach, E. 1996. Non-symmetric scattering in light transport algorithms. In *Rendering Techniques 96*, X. Pueyo and P. Schrder, Eds., Eurographics. Springer Vienna, 81–90.