Содержание

Предисловие 12

Глава 1 ❖ Введение в GLSL 18

Введение 18
Использование загрузчика функций для доступа к новейшей функциональности OpenGL 21
Использование GLM для математических вычислений 25
Использование типов GLM для передачи данных в OpenGL 26
Определение версий GLSL и OpenGL 27
Компиляция шейдера 29
Компоновка шейдерной программы 33
Передача данных в шейдер с использованием вершинных атрибутов и вершинных буферных объектов 36
Получение списка активных атрибутов и их индексов 45
Передача данных в шейдер с использованием uniform-переменных 48
Получение списка активных uniform-переменных 51
Использование uniform-блоков и uniform-буферов 53
Получение отладочных сообщений 59
Создание класса C++, представляющего шейдерную программу 62
Рассеянное отражение с единственным точечным источником света 68

Глава 2 ❖ Основы шейдеров GLSL 66

Введение 66
Фоновый, рассеянный и отраженный свет 73
Использование функций в шейдерах 80
Реализация двустороннего отображения 83
Реализация модели плоского затенения 87
Использование подпрограмм для выбора функциональности в шейдере 89
Отбрасывание фрагментов для получения эффекта решетчатой поверхности 94

Глава 3 ❖ Освещение, затенение и оптимизация 98

Введение 98
Освещение несколькими точечными источниками света 98
Освещение источником направленного света 101
Пофрагментное вычисление освещенности для повышения реализма 104
Использование вектора полупути для повышения производительности 107
Имитация узконаправленных источников света 110
Придание изображению «мультяшного» вида 113
Имитация тумана 116
Настройка проверки глубины 119

Глава 4 ❖ Текстуры 122

Введение 122
Наложение двухмерной текстуры 123
Наложение нескольких текстур 128

Использование карт прозрачности для удаления пикселей ... 131
Использование карт нормалей ... 134
Имитация отражения с помощью кубической текстуры ... 140
Имитация преломления с помощью кубической текстуры ... 147
Наложение проецируемой текстуры ... 152
Отображение в текстуру ... 157
Использование объектов-семплеров ... 162

Глава 5 ❖ Обработка изображений и приемы работы с экранным пространством ... 165

Введение ... 165
Применение фильтра выделения границ ... 166
Применение фильтра размытия по Гауссу ... 172
Преобразование диапазона яркостей HDR с помощью тональной компрессии ... 179
Эффект размытости на границах ярких участков ... 184
Повышение качества изображения с помощью гамма-коррекции ... 189
Сглаживание множественной выборкой ... 192
Отложенное освещение и затенение ... 197
Реализация порядконезависимой прозрачности ... 203

Глава 6 ❖ Использование геометрических шейдеров и шейдеров тесселяции ... 215

Введение ... 215
Отображение точечных спрайтов с помощью геометрического шейдера ... 220
Наложение каркаса на освещенную поверхность ... 225
Рисование линий силуэта с помощью геометрического шейдера ... 233
Тесселяция кривой ... 242
Тесселяция двухмерного прямоугольника ... 247
Тесселяция трехмерной поверхности ... 252
Тесселяция с учетом глубины ... 257

Глава 7 ❖ Тени ... 261

Введение ... 261
Отображение теней с помощью карты теней ... 261
Сглаживание границ теней методом PCF ... 272
Смягчение границ теней методом случайной выборки ... 275
Создание теней с использованием приема теневых объемов и геометрического шейдера ... 282

Глава 8 ❖ Использование шума в шейдерах ... 291

Введение ... 291
Создание текстуры шума с использованием GLM ... 293
Создание бесшовной текстуры шума ... 296
Создание эффекта облаков ... 298
Создание эффекта текстуры древесины ... 300
Создание эффекта разрушения ... 303
Создание эффекта брызг краски ... 305

Создание эффекта изображения в приборе ночного видения 308

Глава 9 ❖ Системы частиц и анимация 312

Введение 312
Анимация поверхности смещением вершин 313
Создание фонтана частиц 316
Создание системы частиц с использованием трансформации с обратной связью 322
Создание системы частиц клонированием 331
Имитация пламени с помощью частиц 334
Имитация дыма с помощью частиц 337

Глава 10 ❖ Вычислительные шейдеры 340

Введение 340
Реализация системы частиц с помощью вычислительного шейдера 344
Имитация полотнища ткани с помощью вычислительного шейдера 348
Определение границ с помощью вычислительного шейдера 355
Создание фракталов с помощью вычислительного шейдера 360

Предметный указатель 364

Об авторе

Дэвид Вольф (David Wolff) – адъюнкт-профессор факультета «Информатика и вычислительная техника» в университете Пасифик Лютеран (Pacific Lutheran University, PLU). Имеет степень кандидата физико-математических наук и магистра информатики, полученную в университете штата Орегон. Преподает компьютерную графику студентам университета Пасифик Лютеран уже более 10 лет.

О рецензентах

Варфоломей Филипек (Bartłomiej Filipek) – опытный разработчик программного обеспечения, всеми силами старающийся передать свой опыт другим. Вот уже пять лет ведет курс программирования компьютерной графики с применением OpenGL в Ягеллонском университете (Краков, Польша). Кроме того, читает лекции и проводит семинары на тему современных приемов программирования на языке C++ в местных университетах.

Имеет огромный опыт разработки программного обеспечения, в том числе низкоуровневого кода для систем отображения, крупномасштабных программных продуктов, игровых движков, мультимедийных приложений, пользовательских интерфейсов, вычислений на GPU и даже приложений для клинического мониторинга.

Он делится своим опытом программирования в своем блоге по адресу: **http://www.bfilipek.com/**.

Томас Ле Гуэро-Древиллон (Thomas Le Guerroué-Drévillon) – инженер-программист, недавно закончивший обучение. Увлекается математикой и рисованием – ему легко удается совмещать эти два увлечения в компьютерной графике.

Родился во Франции, но приложил все усилия, чтобы воспользоваться возможностями, данными ему его университетом, чтобы учиться и работать по всему миру. Первый международный опыт получил в Эстонии, затем обучался в престижном Корейском институте перспективных научных исследований и технологий (KAIST) в Южной Корее и, наконец, сейчас живет в Канаде.

Даже при том, что математическую базу он получил в университете, опыт владения OpenGL и GLSL приобретался им самостоятельно. Томас уверен, что между документированным API и реализацией шейдеров лежит глубокая пропасть, и без колебаний принялся за рецензирование этого сборника рецептов, который он с радостью приобрел бы, когда только начинал осваивать OpenGL/GLSL.

Доктор Мухаммад Мобин Мования (Dr. Muhammad Mobeen Movania) защитил докторскую диссертацию по теме «Перспективы компьютерной графики и визуализации» в Наньянском технологическом университете, Сингапур. После защиты диссертации приступил к работе в научно-исследовательском институте A-Star (Сингапур) по теме «Дополненная реальность на основе систем виртуальной примерки и имитации движения тканей с использованием GPU и OpenGL». Прежде чем поступить в Наньянский университет, работал младшим разработчиком графики в Data Communication and Control (DCC) Pvt. Ltd. (Карачи, Пакистан). Использовал DirectX и OpenGL для создания интерактивных тактических тренажеров реального времени и динамических комплексных тренажеров. В числе его интересов можно назвать: методы объемной визуализации на основе GPU, приемы использования GPU, физика мягких тканей в режиме реального времени, динамическое наложение теней в режиме реального времени, определение столкновений в режиме реального времени и отклик на них, а также иерархические структуры геометрических данных. Является автором проекта OpenCloth (**http://code.google.com/p/opencloth**), в котором реализовал различные алгоритмы имитации тканей с использованием OpenGL. В его блоге (**http://mmmovania.blogspot.com**) можно

найти большое число полезных советов и хитростей, связанных с компьютерной графикой. В свободное время любит сочинять музыку и играть в сквош.

Доктор Мобин издал несколько статей о компьютерной графике и визуализации в режиме реального времени. Недавно закончил работу над книгой по OpenGL («OpenGL Development Cookbook», изданной в Packt Publishing в 2013 году), где описываются практические рецепты программирования графики с использованием современной версии OpenGL. Его перу принадлежит также глава в другой книге («OpenGL Insights», выпущенной издательством AK Peters/CRC Press в 2012 году).

В настоящее время доктор Мобин работает адъюнкт-профессором в университете DHA Suffa University (Карачи, Пакистан), где занимается взращиванием юных дарований, которые завтра станут выдающимися программистами и исследователями.

Дарио Скарпа (Dario Scarpa) занимается программированием последние 15 лет и не намерен оставлять эту работу. Переходя с одного места на другое, он побывал в шкуре и разработчика, и системного администратора, и создателя видеоигр, и экзаменатора по информатике в университете Салерно (Италия). При этом он осуществлял руководство разработкой проекта Zodiac для компании Adventure Productions, интернет-магазина, специализирующегося на распространении приключенческих игр. В то время когда Дарио занимался рецензированием этой книги, он параллельно готовился получить степень магистра со своей дипломной работой по теме компьютерной графики, где демонстрировал обширные возможности – как нетрудно догадаться – OpenGL.

Джавед Раббани Шах (Javed Rabbani Shah) имеет квалификацию инженера-электромеханика, полученную в инженерно-технологическом университете (Лахор, Пакистан) в 2004 году. Свою профессиональную карьеру он начинал в Delta Indus Systems (ныне Vision Master Inc.), где плотно занимался системами статистического контроля процессов, применяя такие технологии, как обработка изображений, трехмерное машинное зрение и динамически перепрограммируемые микросхемы. Затем в 2007 году перешел в подразделение Embedded Systems компании Mentor Graphics, где получил возможность поработать с системой реального времени Nucleus+, микропрограммами USB 2.0, WebKit и OpenGL ES 2.0. Он стал инициатором внедрения OpenGL ES 2.0 в кросс-платформенный движок пользовательского интерфейса Inflexion 3D компании Mentor.

В настоящее время Джавед работает в Saffron Digital, в центре Лондона, где занимается проблемами кросс-платформенной защиты видеофайлов от воспроизведения и технологиями DRM и UltraViolet.

В свободное время изучает смежные технологии, такие как OpenGL ES 3.0 и OpenCL.

www.PacktPub.com

Файлы поддержки, электронные книги, скидки и многое другое

Файлы поддержки для данной книги можно найти на сайте издательства **www.PacktPub.com**.

Знаете ли вы, что издательство Packt предлагает электронные версии всех выпускаемых им книг в форматах PDF и ePub? Приобрести электронные версии книг можно на сайте **www.PacktPub.com**, а при покупке печатной книги вы получите скидку на ее электронную копию. За дополнительной информацией обращайтесь по адресу: **service@packtpub.com**.

На сайте **www.PacktPub.com** можно также найти множество технических статей, подписаться на бесплатные новостные письма и получить исключительные скидки на печатные и электронные книги издательства Packt.

http://PacktLib.PacktPub.com

Вам нужна возможность быстро находить ответы на свои вопросы? К вашим услугам PacktLib – электронная библиотека издательства Packt. Здесь вы получите доступ ко множеству электронных книг.

Что дает подписка?

- Неограниченную возможность поиска по всем книгам издательства Packt.
- Копирование, печать и установку закладок.
- Быстрый доступ к содержимому с помощью веб-браузера.

Свободный доступ для обладателей учетных записей Packt

Если у вас есть учетная запись на сайте **www.PacktPub.com**, вы сможете использовать ее для доступа к PacktLib прямо сегодня и просмотреть девять книг бесплатно. Просто воспользуйтесь своей учетной записью для входа в библиотеку.

Предисловие

Язык шейдеров OpenGL (OpenGL Shading Language, GLSL) в настоящее время является фундаментальной основой и неотъемлемой частью программирования с использованием библиотеки OpenGL. Его применение дает беспрецедентную гибкость и широту возможностей за счет превращения прежде фиксированного конвейера обработки графики в программируемый. Благодаря GLSL можно пользоваться мощью графического процессора (Graphics Processing Unit, GPU) для реализации улучшенных приемов отображения и даже для произвольных вычислений. Версия GLSL 4.x дает программистам еще более широкие возможности управления GPU, чем когда-либо, благодаря введению новых видов шейдеров, таких как шейдеры тесселяции (tessellation shaders) и вычислительные шейдеры.

В этой книге мы рассмотрим весь спектр приемов программирования на GLSL. Начав с базовых видов шейдеров – вершинных и фрагментных, мы пройдем путь от простых до сложных приемов. Мы покажем множество практических примеров – от наложения текстур, воспроизведения теней и обработки изображений до применения искажений и манипуляций системами частиц – чтобы дать инструменты, которые вам понадобятся при использовании GLSL в ваших проектах. Мы также расскажем, как пользоваться геометрическими шейдерами, шейдерами тесселяции и совсем недавно появившимися в GLSL вычислительными шейдерами. С их помощью вы сможете задействовать GPU для решения самых разных задач, даже тех, что никак не связаны с формированием изображений. С помощью геометрических шейдеров и шейдеров тесселяции можно выполнять геометрические построения, а с помощью вычислительных – произвольные вычисления на GPU.

Для тех, кто впервые приступает к изучению GLSL, лучше читать эту книгу по порядку, начав с главы 1. Рецепты даются в порядке увеличения их сложности. Те же, кто уже имеет опыт использования GLSL, могут выбирать рецепты в произвольном порядке. В большинстве своем рецепты не зависят друг от друга, но есть и такие, которые ссылаются на другие рецепты. Введение к каждой главе содержит важную информацию о рассматриваемой теме, поэтому, возможно, вам также стоит прочитать и вводные разделы.

Версия GLSL 4.x превращает программирование с использованием OpenGL в настоящую забаву. Я искренне надеюсь, что эта книга пригодится вам и вы будете пользоваться ее рецептами при работе над своими проектами. Я верю, что программирование в OpenGL и GLSL понравится вам так же, как мне, и что описываемые здесь приемы вдохновят вас на создание красивой графики.

О чем рассказывается в этой книге

Глава 1 «Введение в GLSL» описывает этапы, которые нужно пройти, чтобы скомпилировать, скомпоновать и запустить шейдеры GLSL в программе OpenGL. В ней также рассказывается, как передавать данные в шейдеры с помощью переменных-атрибутов и uniform-переменных и как с помощью библиотеки GLM

осуществлять математические вычисления. Каждой современной программе на основе OpenGL требуется загрузчик функций. В этой главе мы расскажем, как пользоваться GLLoadGen, относительно новым и простым в использовании генератором загрузчиков OpenGL.

Глава 2 «Основы шейдеров GLSL» знакомит с основами программирования на GLSL на примере вершинных шейдеров. В этой главе вы увидите примеры основных приемов программирования шейдеров, таких как алгоритмы освещения ADS (ambient – фоновое, diffuse – рассеянное и specular – глянцевые блики), двустороннее окрашивание и плоское окрашивание. В ней также демонстрируются примеры основных концепций языка GLSL, таких как функции и подпрограммы.

Глава 3 «Освещение, затенение и оптимизация» знакомит с расширенными приемами. Основное внимание в ней будет уделено фрагментным шейдерам. Здесь вы познакомитесь с такими приемами, как освещение точечным источником света, пофрагментное отображение, придание изображению «мультяшного» вида, воспроизведение эффекта тумана, и др. Мы также обсудим несколько простых оптимизаций, которые помогут вам ускорить работу ваших шейдеров.

Глава 4 «Текстуры» представляет собой обобщенное введение в приемы использования текстур в шейдерах GLSL. Текстуры могут использоваться с самыми разными целями, помимо самого простого наложения изображения на поверхность. В этой главе мы познакомим вас с основами использования одной или более двухмерных текстур, а также с различными другими приемами наложения текстур, включая альфа-наложение, наложение по нормалям, кубическое наложение, проекционное наложение и отображение в текстуру. Мы затронем семплеры, относительно новую особенность, которая помогает отделить параметры выборки из текстур от самих текстур.

Глава 5 «Обработка изображений и приемы работы с экранным пространством» описывает типичные способы постобработки созданных изображений и другие приемы работы с экранным пространством. Постобработка изображения является важнейшим элементом современных игровых движков и других средств отображения трехмерной графики. В этой главе рассказывается, как реализовать некоторые из наиболее типичных способов постобработки, таких как создание размытости на границах ярких участков, гамма-коррекция и выделение границ. Здесь также рассматриваются некоторые приемы обработки экранного пространства, такие как отложенное освещение и затенение, сглаживание множественной выборкой и порядко-независимой прозрачности.

Глава 6 «Использование геометрических шейдеров и шейдеров тесселяции» охватывает приемы использования новых видов шейдеров. В этой главе вы познакомитесь с основными функциональными возможностями этих шейдеров и получите представление о том, как ими пользоваться. Здесь рассказывается о таких приемах, как генерация точечных спрайтов геометрическим шейдером, выделение силуэтов, тесселяция с учетом глубины, поверхности Безье и многих других.

Глава 7 «Тени» знакомит с основными приемами отображения теней в реальном масштабе времени. Эта глава включает рецепты, реализующие два основных при-

ема: карты теней и теневые объемы. Мы охватим типичные приемы сглаживания карт теней, а также покажем, как использовать геометрический шейдер для воспроизведения теневых объемов.

Глава 8 «Использование шума в шейдерах» охватывает использование градиентного шума Перлина для создания разнообразных эффектов. Первый рецепт показывает, как создать широкое разнообразие текстур шума с применением GLM (мощной математической библиотеки). Затем мы перейдем к рецептам, использующим подобные текстуры для создания таких эффектов, как текстура древесины, облака, рассыпание, брызги краски и статические разряды.

Глава 9 «Системы частиц и анимация» описывает приемы создания систем частиц. Здесь мы увидим, как с помощью системы частиц воспроизвести имитацию взрыва, облака дыма и брызг воды. В этой главе мы также будем использовать одну особенность OpenGL, которая называется трансформацией с обратной связью, чтобы увеличить производительность алгоритма за счет передачи обновления положения частиц графическому процессору GPU.

Глава 10 «Вычислительные шейдеры» знакомит с несколькими приемами использования одной из новейших особенностей в OpenGL – вычислительного шейдера. Поддержка вычислительных шейдеров открывает широкие возможности для выполнения универсальных вычислений на GPU с поддержкой OpenGL. В этой главе мы обсудим, как использовать вычислительные шейдеры для моделирования частиц, облаков, выделения границ и генерации фрактальной текстуры. К концу этой главы читатель получит хорошее представление о том, как организовать произвольные математические вычисления с помощью вычислительных шейдеров.

Что потребуется для работы с книгой

Рецепты в этой книге демонстрируют использование некоторых новейших особенностей OpenGL 4.x. Поэтому для их опробования вам потребуются графическая аппаратная подсистема (графическая карта или встроенный GPU) и драйверы, поддерживающие, как минимум, версию OpenGL 4.3. Если вы не знаете, какая версия OpenGL поддерживается вашим окружением, воспользуйтесь какой-нибудь из доступных утилит, которая поможет вам получить необходимую информацию. К таким утилитам, например, относится GLview от Realtech VR, доступная по адресу: **http://www.realtech-vr.com/glview/**. Для ОС Windows или Linux всегда можно найти самые свежие драйверы. К сожалению, драйверы для MacOS X часто выходят с запаздыванием, и если вы пользуетесь этой операционной системой, вам, возможно, придется подождать. На момент написания этих строк последняя версия MacOS X (10.9 Mavericks) поддерживала только OpenGL 4.1.

Помимо современных драйверов OpenGL, вам также потребуются:

- компилятор C++. В ОС Linux часто уже имеется установленный пакет GNU Compiler Collection (gcc, g++ и т. д.), но если это не так, его нетрудно установить с помощью менеджера пакетов. В Windows с успехом можно исполь-

зовать Microsoft Visual Studio, но если у вас нет своей копии этой системы разработки, отличным вариантом для вас может стать компилятор MinGW (доступен по адресу: **http://mingw.org/**);
- библиотека GLFW версии 3.0 или выше, доступная по адресу: **http://www.glfw.org/**. Эта библиотека обеспечивает возможность создания контекстов OpenGL, поддержку окон и событий ввода от пользователя;
- библиотека GLM версии 0.9.4 или выше, доступная по адресу: **http://glm.g-truc.net/**. Эта библиотека обеспечивает поддержку математических вычислений и содержит реализации классов матриц, векторов, часто используемых преобразований, функций шума и многое другое.

Кому адресована эта книга

В центре внимания этой книги находится язык программирования шейдеров OpenGL (OpenGL Shading Language, GLSL). Соответственно, мы не будем тратить времени на обсуждение основ программирования с использованием OpenGL. Далее в этой книге я буду предполагать, что читатель уже имеет некоторый опыт программирования в OpenGL и понимает основные концепции трехмерной графики, такие как координаты модели, координаты представления, координаты отсечения, перспективные преобразования и другие виды преобразований. Однако здесь не делается никаких предположений о навыках программирования шейдеров, поэтому, если вы только начинаете осваивать GLSL, данная книга может послужить для вас неплохой отправной точкой.

Если вы имеете опыт использования OpenGL и стремитесь побольше узнать о программировании на GLSL, тогда эта книга для вас. Даже если вы имеете некоторый опыт программирования шейдеров, вы наверняка найдете здесь ценные для вас рецепты. Мы охватим широкий диапазон приемов, от простых до сложных, использующих новейшие особенности OpenGL (такие как вычислительные шейдеры и шейдеры тесселяции). И даже программисты с большим опытом использования GLSL, стремящиеся побольше узнать об этих новейших особенностях, также найдут эту книгу полезной.

Проще говоря, эта книга адресована программистам, знакомым с основами трехмерной графики в OpenGL и заинтересованным в изучении языка GLSL или желающим получить дополнительные сведения о некоторых новейших особенностях GLSL 4.x.

Соглашения

В этой книге используется несколько разных стилей оформления текста с целью обеспечить визуальное отличие информации разных типов. Ниже приводятся несколько примеров таких стилей оформления и краткое описание их назначения.

Программный код в тексте, имена таблиц баз данных, имена папок, имена файлов, расширения файлов, пути в файловой системе, ввод пользователя и ссылки в Twitter оформляются, как показано в следующем предложении: «Число рабочих групп определяется параметром `glDispatchCompute`.».

Блоки программного кода оформляются так:

```
void main()
{
    Color = VertexColor;
    gl_Position = RotationMatrix * vec4(VertexPosition,1.0);
}
```

Когда нам потребуется привлечь ваше внимание к определенному фрагменту в блоке программного кода, мы будем выделять его жирным шрифтом:

```
void main()
{
    Color = VertexColor;
    gl_Position = RotationMatrix * vec4(VertexPosition,1.0);
}
```

Ввод и вывод в командной строке будут оформляться так:

```
Active attributes:
1    VertexColor (vec3)
0    VertexPosition (vec3)
```

Новые термины и **важные определения** будут выделяться в обычном тексте жирным.

Так будут оформляться предупреждения и важные примечания.

Так будут оформляться советы и рекомендации.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезными.

Вы можете написать отзыв прямо на нашем сайте **www.dmkpress.com**, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте **www.dmkpress.com** или **www.дмк.рф** в разделе Читателям – Файлы к книгам.

Также все исходные тексты для всех рецептов в этой книге доступны в GitHub по адресу: **https://github.com/daw42/glslcookbook**.

Цветные иллюстрации для этой книги

Мы бесплатно предоставляем файл PDF с цветными иллюстрациями – скриншотами/диаграммами, что приводятся в этой книге. Цветные иллюстрации помогут вам лучше понять, что происходит на экране. Скачать их можно на сайте **www.dmkpress.com** или **www.дмк.рф** в разделе Читателям – Файлы к книгам.

Ошибки и опечатки

Мы тщательно проверяем содержимое наших книг, но от ошибок никто не застрахован. Если вы найдете ошибку в любой из наших книг – в тексте или в программном коде, мы будем весьма признательны, если вы сообщите нам о ней. Тем самым вы оградите других читателей от разочарований и поможете улучшить последующие версии этой книги.

Если вы найдёте какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и «Packt» очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты **dmkpress@gmail.com** со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Вопросы

Если у вас появились какие-либо вопросы, связанные с нашими книгами, присылайте их по адресу **dmkpress@gmail.com**, а мы приложим все усилия, чтобы ответить на них.

Глава 1

Введение в GLSL

В этой главе описываются следующие рецепты:

- использование загрузчика функций для доступа к новейшим возможностям OpenGL;
- использование GLM для математических вычислений;
- определение версий GLSL и OpenGL;
- компиляция шейдера;
- компоновка шейдерной программы;
- передача данных в шейдер с использованием переменных-атрибутов и буферов;
- получение списка активных атрибутов и их индексов;
- передача данных в шейдер с использованием uniform-переменных;
- получение списка активных uniform-переменных;
- использование uniform-блоков и uniform-буферов;
- получение отладочных сообщений;
- создание класса C++, представляющего шейдерную программу.

Введение

Язык программирования шейдеров OpenGL (OpenGL Shading Language, GLSL) версии 4 дает беспрецедентные возможности и гибкость программистам, заинтересованным в создании современных, интерактивных графических программ. Он позволяет легко и просто использовать мощь современных **графических процессоров (Graphics Processing Unit, GPU)**, предоставляя простые, но мощные языковые конструкции и прикладной программный интерфейс (API). Разумеется, первым шагом к использованию GLSL является создание программы на основе последней версии OpenGL API. Программы на языке GLSL не являются полностью самостоятельными; они должны входить в состав более крупных программ на основе библиотеки OpenGL. В этой главе мы познакомимся с несколькими советами и приемами, которые помогут создать и запустить простую программу. Но перед этим рассмотрим некоторые теоретические выкладки.

Язык шейдеров OpenGL

В настоящее время язык GLSL является фундаментальной и неотъемлемой частью OpenGL API. Забегая вперед, отметим, что любая программа, написанная с привлечением OpenGL API, внутри использует одну или более программ на язы-

ке GLSL. Такие «мини-программы» часто называют **шейдерными программами (shader programs)**. Обычно шейдерная программа состоит из нескольких компонентов, называемых **шейдерами (shaders)**. Каждый шейдер выполняется в рамках отдельного этапа в общем конвейере OpenGL. Каждый шейдер выполняется на GPU и, как можно заключить из названия[1], реализует (обычно) алгоритм, так или иначе связанный с эффектами освещения и затенения в изображении. Однако шейдеры способны на большее, чем просто воспроизводить эффекты освещения и затенения. С их помощью можно также воспроизводить анимацию, выполнять тесселяцию (tessellation) и даже производить универсальные математические вычисления.

Вопросам использования GPU (часто с использованием специализированных API, таких как CUDA или OpenCL) для выполнения универсальных вычислений, например в физике жидкостей и газов, молекулярной динамике, криптографии и т. д., посвящена целая область исследований с названием **GPGPU (General Purpose Computing on Graphics Processing Units – универсальные вычисления на графических процессорах)**. Благодаря появлению вычислительных шейдеров в версии OpenGL 4.3 мы теперь можем выполнять аналогичные вычисления в рамках OpenGL.

Шейдерные программы предназначены для непосредственного выполнения на GPU и действуют параллельно. Например, фрагментный шейдер может вызываться для каждого пикселя, при этом все пиксели могут обрабатываться одновременно, в отдельных потоках выполнения на GPU. Число процессоров в графической карте определяет, сколько шейдеров может выполняться одновременно. Это обстоятельство делает шейдерные программы чрезвычайно эффективными, а программист получает в свое распоряжение простой API для реализации вычислений с высокой степенью параллелизма.

Вычислительная мощность современных графических процессоров потрясает. В табл. 1.1 приводится число шейдерных процессоров, имеющееся в некоторых моделях графических карт серии GeForce, выпускаемых компанией NVIDIA (источник: **http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units**).

Таблица 1.1. Сравнение графических карт NVIDIA

Модель	Число шейдерных процессоров
GeForce GTS 450	192
GeForce GTX 480	480
GeForce GTX 780	2304

Шейдерные программы предназначены для замены части архитектуры OpenGL, которую называют **конвейером с фиксированной функциональностью (fixed-function pipeline)**. В OpenGL до версии 2.0 алгоритмы отображения были

[1] Дословно название *shader* переводится на русский язык как «программа построения теней». – *Прим. перев.*

«жестко зашиты» в функциональный конвейер и имели весьма ограниченные возможности настройки. Этот алгоритм отображения, действующий по умолчанию, составлял основу конвейера с фиксированной функциональностью. Когда нам как программистам требовалось реализовать более сложные или более реалистичные эффекты, мы использовали различные трюки, чтобы хоть как-то повысить гибкость конвейера с фиксированной функциональностью. Появление поддержки языка GLSL дало возможность заменять «жестко зашитую» функциональность своим программным кодом, написанным на GLSL, и помогло нам получить дополнительную гибкость и возможности. Более подробно о конвейере с программируемой функциональностью рассказывается в главе 2 «Основы шейдеров GLSL».

Фактически последние (основные) версии OpenGL не только дают такую возможность, но даже требуют, чтобы шейдерные программы входили в состав любых программ, использующих OpenGL. Старый конвейер с фиксированной функциональностью был объявлен устаревшим, и предпочтение отдано новому конвейеру с программируемой функциональностью, ключевым элементом которого являются шейдеры, написанные на GLSL.

Профили – базовый и совместимости

В версии OpenGL 3.0 появилась **модель определения устаревшей функциональности (deprecation model)**, обеспечивающая возможность постепенного устранения функций из спецификации OpenGL. Как предполагается, функции или особенности, объявленные устаревшими, будут удаляться в будущих версиях OpenGL. Например, непосредственный режим отображения с использованием `glBegin/glEnd` был объявлен устаревшим в версии 3.0 и убран в версии 3.1.

В OpenGL 3.2 для поддержки обратной совместимости была добавлена концепция **профилей совместимости (compatibility profiles)**. Программист, пишущий код для какой-то определенной версии OpenGL (из которой были удалены устаревшие функции), может использовать так называемый **базовый профиль (core profile)**. Любой, кто пожелает поддерживать совместимость с устаревшей функциональностью, может использовать **профиль совместимости (compatibility profile)**.

Кому-то может показаться странным, что существует также понятие контекста **опережающей совместимости (forward compatible)**, или совместимости снизу вверх, которое несколько отличается от понятий базового профиля и профиля совместимости. Контекст опережающей совместимости, как считается, просто указывает, что вся устаревшая функциональность была удалена. Иными словами, если контекст объявлен совместимым снизу вверх, это означает, что он включает только функции базового профиля, за исключением функций, объявленных устаревшими. Некоторые оконные API предоставляют возможность выбрать статус опережающей совместимости наряду с профилем.

Шаги, которые требуется выполнить для выбора базового профиля или профиля совместимости, зависят от API оконной системы. Например, в GLFW выбрать контекст опережающей совместимости и базовый профиль 4.3 можно с помощью следующего кода:

```
glfwWindowHint( GLFW_CONTEXT_VERSION_MAJOR, 4 );
glfwWindowHint( GLFW_CONTEXT_VERSION_MINOR, 3 );
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

GLFWwindow *window = glfwCreateWindow(640, 480, "Title",
      NULL, NULL);
```

Все программы, что приводятся в данной книге, проектировались с учетом опережающей совместимости с базовым профилем OpenGL 4.3.

Использование загрузчика функций для доступа к новейшей функциональности OpenGL

В ОС Windows OpenGL **ABI (application binary interface – двоичный прикладной интерфейс)** был заморожен в версии OpenGL 1.1. К сожалению, для Windows-разработчиков это означает, что они не могут использовать функции новейших версий OpenGL непосредственно. Вместо этого они должны получать доступ к таким функциям, приобретая указатели на них во время выполнения. Получить указатели на функции несложно, но для этого требуется выполнить дополнительную рутинную работу и написать дополнительный код, захламляющий программу. Кроме того, обычно в состав Windows включается стандартный заголовочный файл OpenGL `gl.h`, который также соответствует версии OpenGL 1.1. На вики-странице OpenGL отмечается, что компания Microsoft не планирует обновлять файлы `gl.h` и `opengl32.lib`, поставляемые вместе с их компиляторами. К счастью, другие позаботились о том, чтобы предоставить обновленные заголовочные файлы, а также библиотеки, которые автоматически выполняют рутинную работу и обеспечивают прозрачный доступ к необходимым указателям на функции. Существует несколько библиотек, обеспечивающих такого рода поддержку. Одной из старейших и наиболее широко используемых является библиотека **GLEW (OpenGL Extension Wrangler)**. Однако она имеет несколько серьезных недостатков, снижающих ее ценность и сделавших ее недостаточной для моих целей, когда я писал эту книгу. Во-первых, на момент написания этих слов она некорректно поддерживала базовые профили, тогда как в этой книге я хотел сосредоточиться только на новейшей функциональности, без устаревших функций. Во-вторых, в ее состав входит огромный заголовочный файл, включающий все, что когда-либо входило в состав OpenGL. Было бы предпочтительнее иметь более короткий заголовочный файл, включающий только те функции, которые можно использовать. Наконец, библиотека GLEW распространяется в виде исходных текстов, которые нужно компилировать отдельно и компоновать с проектом. Часто бывает предпочтительнее иметь загрузчик, который можно включить в проект простым добавлением файлов с исходным кодом и компилировать его непосредственно в выполняемый файл, избежав необходимости поддерживать дополнительную зависимость.

В данном рецепте будет использоваться утилита **OpenGL Loader Generator** (GLLoadGen), доступная по адресу: **https://bitbucket.org/alfonse/glloadgen/**

wiki/Home. Это очень гибкий и эффективный инструмент, решающий все три проблемы, описанные выше. Он поддерживает базовые профили, может генерировать заголовочные файлы, включающие только то, что необходимо, и генерирует лишь пару файлов (файл с исходным кодом и заголовочный файл), которые можно включить непосредственно в проект.

Подготовка

Чтобы воспользоваться утилитой GLLoadGen, необходима поддержка **Lua**. Lua – это легковесный встраиваемый язык сценариев, доступный практически для любых платформ. Двоичные файлы можно найти по адресу: **http://luabinaries.sourceforge.net**, а полностью готовый к установке комплект для Windows (LuaForWindows) можно загрузить по адресу: **https://code.google.com/p/luaforwindows**.

Загрузите дистрибутив GLLoadGen по адресу: **https://bitbucket.org/alfonse/glloadgen/downloads**. Файл дистрибутива сжат архиватором 7zip. Этот архиватор установлен не везде, поэтому вам, возможно, потребуется установить его, загрузив утилиту 7zip по адресу: **http://7-zip.org/**. Распакуйте дистрибутив в каталог по выбору. Так как утилита GLLoadGen написана на Lua, ее не нужно компилировать – сразу после распаковки дистрибутива она готова к использованию.

Как это делается...

Первый шаг – сгенерировать заголовочный файл и файл с исходным кодом для выбранной версии OpenGL и профиля. В данном примере мы сгенерируем файлы для базового профиля OpenGL 4.3. Сгенерированные файлы можно скопировать в проект и скомпилировать их вместе с исходным кодом проекта:

1. Чтобы сгенерировать файлы, перейдите в каталог, куда был распакован дистрибутив GLLoadGen, и запустите команду `GLLoadGen` со следующими аргументами:

   ```
   lua LoadGen.lua -style=pointer_c -spec=gl -version=4.3 \
   -profile=core core_4_3
   ```

2. После выполнения предыдущего шага должны появиться два файла: `gl_core_4_3.c` и `gl_core_4_3.h`. Скопируйте эти файлы в свой проект и включите `gl_core_4_3.c` в сборку. В своем программном коде, где требуется получить доступ к функциям OpenGL, подключите `gl_core_4_3.h`. Однако этого недостаточно – чтобы инициализировать указатели на функции, требуется также вызвать функцию `ogl_LoadFunctions`. Где-нибудь, сразу после создания контекста GL (обычно в функции инициализации) и перед вызовами любых функций OpenGL, вставьте следующий код:

   ```
   int loaded = ogl_LoadFunctions();
   if(loaded == ogl_LOAD_FAILED) {
     // Освободить контекст и прервать выполнение
     return;
   }
   ```

```
int num_failed = loaded - ogl_LOAD_SUCCEEDED;
printf("Number of functions that failed to load: %i.\n",
       num_failed);
```

Вот и все!

Как это работает...

Команда `lua`, что вызывается на шаге 1, сгенерирует пару файлов – заголовочный файл и файл с исходным кодом. Заголовочный файл содержит определения прототипов всех выбранных функций OpenGL и переопределяет их как указатели на функции, а также определяет все константы OpenGL. Файл с исходным кодом содержит реализацию процедуры инициализации указателей на функции и несколько вспомогательных функций. Мы можем подключить `gl_core_4_3.h` к любому файлу, где необходимы объявления прототипов функций OpenGL, благодаря чему все точки входа в функции будут доступны на этапе компиляции. На этапе выполнения `ogl_LoadFunctions()` инициализирует все имеющиеся указатели на функции. Если какие-то функции не удастся загрузить, число таких функций можно определить вычитанием, как показано в шаге 2. Если требуемая функция отсутствует в выбранной версии OpenGL, код не скомпилируется, потому что в заголовочном файле будут присутствовать определения прототипов функций только для выбранной версии OpenGL и профиля.

Полное описание всех аргументов командной строки для GLLoadGen можно найти по адресу: **https://bitbucket.org/alfonse/glloadgen/wiki/Command_Line_Options**. Предыдущий пример показывает наиболее типичные настройки, но существует еще масса других, обеспечивающих этому инструменту высокую гибкость.

Теперь, получив пару файлов, мы больше не зависим от утилиты GLLoadGen, и наша программа может быть скомпилирована без ее участия. Это большое преимущество перед другими инструментами, такими как GLEW.

И еще...

Утилита GLLoadGen имеет еще несколько интересных и весьма полезных особенностей. С ее помощью можно генерировать код, более дружественный для C++, управлять расширениями и генерировать файлы, не требующие вызова функции инициализации.

Создание загрузчика на C++

Утилита GLLoadGen способна также генерировать заголовочный файл и файл с исходным кодом на C++. Добиться этого можно, передав параметр `-style`. Например, чтобы сгенерировать файлы на языке C++, нужно передать параметр `-style=pointer_cpp`, как в следующем примере:

```
lua LoadGen.lua -style=pointer_cpp -spec=gl -version=4.3 \
-profile=core core_4_3
```

Эта команда сгенерирует файлы gl_core_4_3.cpp и gl_core_4_3.hpp. Все определения функций OpenGL в этих файлах будут помещены в пространство имен gl::, и из их имен будет убран префикс gl (или GL). Например, чтобы вызвать функцию glBufferData, вам придется использовать следующий синтаксис:

```
gl::BufferData(gl::ARRAY_BUFFER, size, data, gl::STATIC_DRAW);
```

Загрузка указателей функций также немного отличается. Возвращаемое значение на этот раз является объектом, а не простым целым числом, и функция LoadFunctions теперь находится в пространстве имен gl::sys.

```
gl::exts::LoadTest didLoad = gl::sys::LoadFunctions();

if(!didLoad) {
    // освободить ресурсы (разрушить контекст) и прервать выполнение.
    return;
}
printf("Number of functions that failed to load: %i.\n",
  didLoad.GetNumMissing());
```

Автоматическая загрузка

GLLoadGen поддерживает автоматическую инициализацию указателей на функции. Она включается передачей значения noload_c или noload_cpp в параметре -style. В этом случае отпадает необходимость вызывать функцию инициализации ogl_LoadFunctions. Указатели загружаются автоматически, при вызове первой же функции. Возможно, это удобно, но такой режим работы влечет за собой небольшие накладные расходы на инициализацию.

Использование расширений

Утилита GLLoadGen не поддерживает расширения автоматически – их необходимо передавать явно в параметрах командной строки. Например, чтобы задействовать расширения ARB_texture_view и ARB_vertex_attrib_binding, можно воспользоваться следующей командой:

```
lua LoadGen.lua -style=pointer_c -spec=gl -version=3.3 \
-profile=core core_3_3 \
-exts ARB_texture_view ARB_vertex_attrib_binding
```

В параметре -exts передается список расширений, перечисленных через пробел. GLLoadGen поддерживает также возможность загружать списки расширений из файлов (с помощью параметра -extfile), а на веб-сайте проекта можно найти несколько файлов с расширениями.

GLLoadGen может также проверять наличие расширений во время выполнения. Подробности смотрите на вики-странице GLLoadGen.

См. также

GLEW, более старый и широко используемый загрузчик расширений и менеджер расширений, доступный по адресу: **glew.sourceforge.net**.

Использование GLM для математических вычислений

В основе компьютерной графики лежит математика. В ранних версиях OpenGL предоставлялась возможность управления преобразованиями и проекциями координат с использованием стандартных матричных стеков (`GL_MODELVIEW` и `GL_PROJECTION`). Однако в последних версиях OpenGL все, что относилось к матричным стекам, было удалено из библиотеки. Поэтому теперь мы должны сами реализовать все необходимое для поддержки матриц преобразований и проекций и затем передавать их в шейдеры. Конечно, можно было бы написать собственные классы матриц и векторов, но многие предпочитают готовые, надежные решения.

Одним из таких решений является библиотека **GLM (OpenGL Mathematics)**, написанная Кристофом Риччио (Christophe Riccio). Она разработана на основе спецификации GLSL, поэтому синтаксис ее функций очень похож на поддержку математических операций в GLSL. Опытные программисты на GLSL без труда освоят библиотеку GLM. Кроме того, для этой библиотеки имеется множество расширений, включая функции, которых очень не хватает в OpenGL, такие как `glOrtho`, `glRotate` или `gluLookAt`.

Подготовка

Поскольку вся библиотека GLM определена исключительно в заголовочных файлах, она устанавливается очень просто. Загрузите дистрибутив с последней версией GLM по адресу: **http://glm.g-truc.net**. Распакуйте архив и скопируйте каталог `glm` куда-нибудь в дерево каталогов, где компилятор будет искать подключаемые заголовочные файлы.

Как это делается...

Чтобы задействовать библиотеку GLM, достаточно просто подключить основной заголовочный файл и заголовочные файлы с расширениями. В этом примере мы подключим расширение, реализующее преобразования матриц:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

После этого программе станут доступны классы GLM из пространства имен `glm`. Ниже демонстрируется, как используются некоторые из них:

```
glm::vec4 position = glm::vec4( 1.0f, 0.0f, 0.0f, 1.0f );
glm::mat4 view = glm::lookAt( glm::vec3(0.0,0.0,5.0),
       glm::vec3(0.0,0.0,0.0),
       glm::vec3(0.0,1.0,0.0) );
glm::mat4 model(1.0f); // Единичная матрица
model = glm::rotate( model, 90.0f, glm::vec3(0.0f,1.0f,0.0) );
glm::mat4 mv = view * model;
glm::vec4 transformed = mv * position;
```

Как это работает...

Библиотека GLM состоит только из заголовочных файлов. То есть вся реализация подключается к программе в виде заголовочных файлов. Ее не требуется компилировать отдельно и затем компоновать с программой – достаточно расположить файлы библиотеки где-нибудь в пути поиска заголовочных файлов, и все!

В примере выше сначала создается объект типа `vec4` (вектор с четырьмя координатами), представляющий позицию в пространстве. Затем вызовом функции `glm::lookAt` создается матрица вида, имеющая размер 4×4. Эта функция действует подобно устаревшей функции `gluLookAt`. Здесь камера устанавливается в точку с координатами (0, 0, 5) и направляется в начало координат, а направление «вверх» совпадает с направлением оси Y. Далее создается матрица модели: сначала переменная `model` инициализируется как единичная матрица (передачей единственного аргумента конструктору) и затем умножается на матрицу поворота вызовом функции `glm::rotate`. Умножение здесь выполняется неявно. Функция `glm::rotate` умножает (справа) свой первый параметр на матрицу поворота, генерируемую функцией. Во втором параметре передается угол поворота (в градусах), а в третьем – ось, вокруг которой осуществляется поворот. Так как перед этой инструкцией переменная `model` хранила единичную матрицу, в конечном итоге она превращается в матрицу поворота на 90° вокруг оси Y.

В заключение умножением переменных `model` и `view` создается матрица модели вида (`mv`), после чего полученная комбинированная матрица используется для преобразования позиции `position`. Обратите внимание, что оператор умножения перегружен, благодаря чему достигается требуемый результат.

И еще...

Не рекомендуется импортировать все пространство имен GLM, как показано ниже:

```
using namespace glm;
```

Это наверняка вызовет множество конфликтов имен. Предпочтительнее импортировать имена по одному, по мере их необходимости. Например:

```
#include <glm/glm.hpp>
using glm::vec3;
using glm::mat4;
```

Использование типов GLM для передачи данных в OpenGL

Библиотека GLM поддерживает непосредственную передачу типов GLM в OpenGL с использованием векторных функций OpenGL (имена которых оканчиваются на `v`). Например, ниже показано, как передать переменную `proj` типа `mat4`:

```
glm::mat4 proj = glm::perspective( viewAngle, aspect, nearDist,
    farDist );
glUniformMatrix4fv(location, 1, GL_FALSE, &proj[0][0]);
```

См. также

- Пакет Qt SDK включает множество классов, поддерживающих операции с векторами/матрицами, и их можно считать отличной альтернативой, если вы уже имеете опыт использования Qt.
- Веб-сайт GLM (**http://glm.g-truc.net**), где можно найти дополнительную документацию и примеры.

Определение версий GLSL и OpenGL

При необходимости поддерживать широкий спектр систем очень важно уметь определять версии OpenGL и GLSL, поддерживаемые текущим драйвером. Сделать это совсем несложно – достаточно воспользоваться всего двумя функциями: `glGetString` и `glGetIntegerv`.

Как это делается...

Следующий фрагмент выведет информацию о версии в `stdout`:

```
const GLubyte *renderer = glGetString( GL_RENDERER );
const GLubyte *vendor = glGetString( GL_VENDOR );
const GLubyte *version = glGetString( GL_VERSION );
const GLubyte *glslVersion =
      glGetString( GL_SHADING_LANGUAGE_VERSION );

GLint major, minor;
glGetIntegerv(GL_MAJOR_VERSION, &major);
glGetIntegerv(GL_MINOR_VERSION, &minor);

printf("GL Vendor            : %s\n", vendor);
printf("GL Renderer          : %s\n", renderer);
printf("GL Version (string)  : %s\n", version);
printf("GL Version (integer) : %d.%d\n", major, minor);
printf("GLSL Version         : %s\n", glslVersion);
```

Как это работает...

Обратите внимание, что существуют два разных способа получить версию OpenGL: с помощью `glGetString` и с помощью `glGetIntegerv`. Первая функция может пригодиться, когда желательно получить информацию в удобочитаемом виде, но ее неудобно использовать для анализа версии в программе, потому что для этого придется проанализировать строку. Строка, возвращаемая вызовом `glGetString(GL_VERSION)`, всегда должна начинаться со старшего и младшего номеров версии, разделенных точкой, при этом младший номер может сопровождаться дополнительным числом – номером сборки, разным для разных производителей. Кроме того, остальная часть строки может содержать дополнительную инфор-

мацию о производителе и включать сведения о выбранном профиле (см. раздел «Введение» в этой главе). Важно отметить, что функцию `glGetIntegerv` можно использовать только в версии OpenGL 3.0 или выше.

При вызове с аргументами `GL_VENDOR` и `GL_RENDERER` функция возвращает дополнительную информацию о драйвере OpenGL. Вызов `glGetString(GL_VENDOR)` вернет название компании-производителя текущей реализации OpenGL. Вызов `glGetString(GL_RENDERER)` вернет название аппаратной платформы (например, ATI Radeon HD 5600 Series). Имейте в виду, что оба аргумента действуют одинаково в разных версиях, поэтому их можно использовать для определения текущей платформы.

Но самое важное для нас в контексте этой книги заключается в том, что вызов `glGetString(GL_SHADING_LANGUAGE_VERSION)` возвращает номер поддерживаемой версии GLSL. Возвращаемая строка должна начинаться со старшего и младшего номеров, разделенных точкой, и, так же как строка, полученная при вызове с аргументом `GL_VERSION`, может включать дополнительную информацию о производителе.

И еще...

Часто бывает полезно узнать, какие расширения поддерживаются текущей реализацией OpenGL. В версиях ниже OpenGL 3.0 можно было получить полный список имен расширений, разделенных пробелами, как показано ниже:

```
GLubyte *extensions = glGetString(GL_EXTENSIONS);
```

Возвращаемая строка могла оказаться очень длинной, и ее анализ мог приводить к ошибкам при невнимательном отношении.

В OpenGL 3.0 стал поддерживаться новый способ, а прежняя функциональность была объявлена устаревшей (и полностью удалена в версии 3.1). Теперь имена расширений индексируются, и их можно запрашивать по индексам. Мы используем версию `glGetStringi` для этого. Например, получить имя расширения с индексом `i` можно так: `glGetStringi(GL_EXTENSIONS, i)`. Ниже демонстрируется фрагмент, который выведет список всех расширений:

```
GLint nExtensions;
glGetIntegerv(GL_NUM_EXTENSIONS, &nExtensions);

for( int i = 0; i < nExtensions; i++ )
      printf("%s\n", glGetStringi( GL_EXTENSIONS, i ) );
```

См. также

- Инструмент GLLoadGen обладает дополнительной поддержкой получения информации о версии и расширениях. Обращайтесь к рецепту «Использование загрузчика функций для доступа к новейшей функциональности OpenGL» и веб-сайту GLLoadGen.

Компиляция шейдера

Сначала нам нужно узнать, как компилируются шейдеры GLSL. Компилятор GLSL встроен непосредственно в библиотеку OpenGL, и шейдеры могут компилироваться только в контексте выполняющейся программы. В настоящее время не существует внешних инструментов для предварительной компиляции шейдеров GLSL и/или шейдерных программ.

Совсем недавно, в версии OpenGL 4.1, была добавлена возможность сохранять скомпилированные шейдеры в файлы, что позволяет программам избежать затрат на их компиляцию за счет загрузки предварительно скомпилированного кода.

Компиляция шейдера заключается в создании объекта шейдера, сохранении в нем исходного кода (в виде строки или множества строк) и вызове метода компиляции. Эта процедура схематически изображена на рис. 1.1.

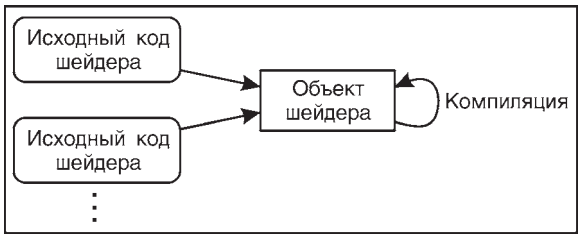


Рис. 1.1 ❖ Процесс компиляции шейдера

Подготовка

Чтобы исследовать процедуру компиляции шейдера, нам нужен простой пример исходного кода. Начнем со следующего простого вершинного шейдера. Сохраните его в файле `basic.vert`.

```
#version 430

in vec3 VertexPosition;
in vec3 VertexColor;

out vec3 Color;

void main()
{
   Color = VertexColor;
   gl_Position = vec4( VertexPosition, 1.0 );
}
```

Для тех, кому интересно знать, что делает этот код: он действует как «мостик» – принимает атрибуты `VertexPosition` и `VertexColor` и передает их фрагментному шейдеру через переменные `gl_Position` и `Color`.

Далее нужно создать каркас программы OpenGL, используя инструментарий, поддерживающий OpenGL. Примерами таких инструментариев могут служить GLFW, GLUT, FLTK, Qt и wxWidgets. В оставшейся части книги я буду предполагать, что вы в состоянии самостоятельно создать простую программу на основе OpenGL с применением своего любимого инструментария. Фактически все инструменты имеют точки для регистрации функций инициализации, обработчиков изменения размеров окна и обработчиков, осуществляющих рисование (которые вызываются при каждом обновлении окна). Для целей данного рецепта нам нужна программа, создающая и инициализирующая контекст OpenGL, она ничего не должна делать, кроме отображения пустого окна OpenGL. Имейте в виду, что вам также потребуется загрузить указатели на функции OpenGL (см. рецепт «Использование загрузчика функций для доступа к новейшей функциональности OpenGL»).

Наконец, добавьте в программу загрузку исходного кода шейдера в массив символов с именем `shaderCode` и не забудьте добавить нулевой символ в конец! Данный пример предполагает, что переменная `shaderCode` указывает на массив типа `GLchar`, завершающийся нулевым символом.

Как это делается...

Чтобы скомпилировать шейдер, требуется выполнить следующие шаги:

1. Создать объект шейдера:

```
GLuint vertShader = glCreateShader( GL_VERTEX_SHADER );
if( 0 == vertShader )
{
  fprintf(stderr, "Error creating vertex shader.\n");
  exit(EXIT_FAILURE);
}
```

2. Скопировать исходный код (возможно, из нескольких мест) в объект шейдера:

```
const GLchar * shaderCode = loadShaderAsString("basic.vert");
const GLchar* codeArray[] = {shaderCode};
glShaderSource( vertShader, 1, codeArray, NULL );
```

3. Скомпилировать шейдер:

```
glCompileShader( vertShader );
```

4. Проверить результат компиляции:

```
GLint result;
glGetShaderiv( vertShader, GL_COMPILE_STATUS, &result );
if( GL_FALSE == result )
{
  fprintf(stderr, "Vertex shader compilation failed!\n");

  GLint logLen;
  glGetShaderiv(vertShader, GL_INFO_LOG_LENGTH, &logLen);
```

```
    if( logLen > 0 )
    {
      char * log = new char[logLen];

      GLsizei written;
      glGetShaderInfoLog(vertShader, logLen, &written, log);

      fprintf(stderr, "Shader log:\n%s", log);
      delete [] log;
    }
  }
```

Как это работает...

Первый шаг – создать объект шейдера вызовом функции `glCreateShader`. Ее аргумент определяет тип шейдера и может иметь одно из следующих значений: `GL_VERTEX_SHADER`, `GL_FRAGMENT_SHADER`, `GL_GEOMETRY_SHADER`, `GL_TESS_EVALUATION_SHADER`, `GL_TESS_CONTROL_SHADER` или (начиная с версии 4.3) `GL_COMPUTE_SHADER`. В данном случае, так как компилируется вершинный шейдер, функции передается аргумент `GL_VERTEX_SHADER`. Функция возвращает значение, которое можно использовать для ссылки на объект вершинного шейдера, иногда это значение называют «дескриптором», или «описателем». В примере это значение сохраняется в переменной `vertShader`. Если во время создания объекта шейдера возникнет ошибка, функция вернет `0`. В этом случае мы выводим соответствующее сообщение и завершаем программу.

Вслед за созданием объекта шейдера выполняется загрузка исходного кода в него вызовом функции `glShaderSource`. Эта функция принимает массив строк, благодаря чему поддерживается возможность компилировать код сразу из множества источников (файлов, строк). Поэтому перед вызовом `glShaderSource` указатель на строку с исходным кодом помещается в массив с именем `sourceArray`. Первый аргумент `glShaderSource` – дескриптор объекта шейдера. Второй – число строк с исходным кодом в массиве. Третий аргумент – указатель на массив строк с исходным кодом. И последний аргумент – массив значений типа `GLint` с длинами строк в предыдущем аргументе-массиве. В данном примере в четвертом аргументе передается значение `NULL`, указывающее, что все строки с исходным кодом оканчиваются нулевым символом. Если бы строки с исходным кодом не оканчивались нулевым символом, тогда мы должны были бы передать в этом аргументе массив значений типа `GLint`. Обратите внимание, что эта функция копирует исходный код во внутреннюю память OpenGL, поэтому память программы, занятую исходным кодом, можно освободить.

Следующий шаг – компиляция исходного кода шейдера. Делается это простым вызовом функции `glCompileShader` и передачей ей дескриптора шейдера, который требуется скомпилировать. Конечно, при наличии ошибок в исходном коде компиляция может потерпеть неудачу, поэтому далее проверяется успех компиляции.

Проверить успех компиляции можно вызовом `glGetShaderiv` – функции, используемой для получения атрибутов объектов шейдеров. В данном случае тре-

буется получить код завершения компиляции, поэтому во втором аргументе передается GL_COMPILE_STATUS. В первом аргументе передается, конечно же, дескриптор объекта шейдера, а в третьем – указатель на целочисленную переменную, куда следует сохранить код. Функция может записать в эту переменную одно из двух значений – GL_TRUE или GL_FALSE, сообщающих об успехе или неуспехе компиляции, соответственно.

Если функция вернет код завершения компиляции GL_FALSE, из журнала шейдера можно извлечь дополнительную информацию с описанием ошибки. С этой целью мы сначала определяем длину записи в журнале вызовом glGetShaderiv с аргументом GL_INFO_LOG_LENGTH и сохраняем ее в переменной logLen. Обратите внимание, что возвращаемое значение длины уже учитывает заключительный нулевой символ. Далее выделяется память для строки и вызовом glGetShaderInfoLog извлекается запись из журнала. Первый параметр этой функции – дескриптор объекта шейдера, второй – размер буфера символов для сохранения записи из журнала, третий – указатель на целочисленную переменную, куда будет записано фактическое число символов (без учета заключительного нулевого символа), скопированных в буфер, и четвертый аргумент – указатель на буфер символов для самой записи из журнала. После извлечения дополнительной информации она выводится в stderr, после чего занятая память освобождается.

И еще...

В предыдущем примере демонстрировалась компиляция только вершинного шейдера, однако существуют шейдеры других типов, в том числе фрагментные шейдеры, геометрические шейдеры и шейдеры тесселяции. Процедуры компиляции разных типов шейдеров почти не отличаются. Единственным важным отличием является аргумент в вызове функции glCreateShader.

Важно также отметить, что компиляция шейдера – это лишь первый шаг. Чтобы создать действующую шейдерную программу, часто требуется скомпилировать хотя бы два шейдера и затем скомпоновать их в объект шейдерной программы. Шаги, связанные с компоновкой, описываются в следующем рецепте.

Удаление объекта шейдера

Объекты шейдеров можно удалять после того, как они станут не нужны, вызовом glDeleteShader. Эта функция освободит память, занятую шейдером, и сделает недействительным дескриптор, ссылающийся на него. Обратите внимание, что если объект шейдера уже подключен к объекту программы (см. рецепт «Компоновка шейдерной программы»), удаление произойдет только после отключения от объекта программы.

См. также

- Рецепт «Компоновка шейдерной программы».

Компоновка шейдерной программы

После компиляции шейдеров и перед тем, как они фактически будут добавлены в конвейер OpenGL, необходимо скомпоновать их в шейдерную программу. Помимо всего прочего, на этапе компоновки производится стыковка входных переменных одного шейдера с выходными переменными другого, а также стыковка входных/выходных переменных шейдеров с соответствующими областями памяти в окружении OpenGL.

Шаги, выполняемые на этапе компоновки (см. рис. 1.2), делают его похожим на этап компиляции: каждый объект шейдера подключается к новому объекту шейдерной программы, и затем этому объекту дается команда выполнить компоновку (перед компоновкой все шейдеры обязательно должны быть скомпилированы).

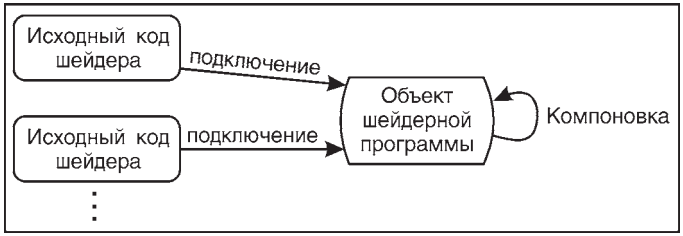


Рис. 1.2 ❖ Процесс компоновки шейдерной программы

Подготовка

В этом рецепте предполагается, что вы уже скомпилировали два объекта шейдеров, дескрипторы которых хранятся в переменных `vertShader` и `fragShader`.

Здесь и в некоторых других рецептах в этой главе будет использоваться следующий фрагментный шейдер:

```
#version 430

in vec3 Color;

out vec4 FragColor;

void main() {
  FragColor = vec4(Color, 1.0);
}
```

Исходный код вершинного шейдера был представлен в предыдущем рецепте «Компиляция шейдера».

Как это делается...

В функции инициализации OpenGL, после компиляции объектов шейдеров, дескрипторы которых хранятся в переменных `vertShader` и `fragShader`, выполняются следующие шаги:

1. Создать объект программы:

```
GLuint programHandle = glCreateProgram();
if( 0 == programHandle )
{
  fprintf(stderr, "Error creating program object.\n");
  exit(1);
}
```

2. Подключить шейдеры к объекту программы:

```
glAttachShader( programHandle, vertShader );
glAttachShader( programHandle, fragShader );
```

3. Скомпоновать программу:

```
glLinkProgram( programHandle );
```

4. Проверить успех компоновки:

```
GLint status;
glGetProgramiv( programHandle, GL_LINK_STATUS, &status );
if( GL_FALSE == status ) {

  fprintf( stderr, "Failed to link shader program!\n" );

  GLint logLen;
  glGetProgramiv(programHandle, GL_INFO_LOG_LENGTH,
         &logLen);
  if( logLen > 0 )
  {
    char * log = new char[logLen];
    GLsizei written;
    glGetProgramInfoLog(programHandle, logLen, &written, log);
    fprintf(stderr, "Program log: \n%s", log);
    delete [] log;
  }
}
```

5. В случае успеха компоновки добавить программу в конвейер OpenGL:

```
else
{
  glUseProgram( programHandle );
}
```

Как это работает...

Сначала вызывается функция `glCreateProgram`, чтобы создать пустой объект программы. Она возвращает дескриптор объекта программы, который мы сохраняем в переменной `programHandle`. Если во время создания объекта программы возникнет ошибка, функция вернет `0`. В этом случае мы выводим соответствующее сообщение и завершаем программу.

Далее выполняется подключение каждого шейдера к объекту программы вызовом функции `glAttachShader`. В первом аргументе ей передается дескриптор объекта программы, а во втором – дескриптор подключаемого объекта шейдера.

Затем производится компоновка программы вызовом функции glLinkProgram, которой в качестве единственного аргумента передается дескриптор объекта программы. Как и в процедуре компиляции, далее проверяется успех процедуры компоновки.

Проверка выполняется вызовом glGetProgramiv. Подобно glGetShaderiv, функция glGetProgramiv позволяет получать значения разных атрибутов шейдерной программы. В данном случае передачей во втором аргументе значения GL_LINK_STATUS запрашивается код завершения компоновки. Код возвращается в переменной, на которую ссылается указатель в третьем аргументе, то есть в переменной status.

Код компоновки может иметь одно из двух значений – GL_TRUE или GL_FALSE, сообщающих об успехе или неуспехе компоновки соответственно. Если возвращается код завершения GL_FALSE, мы извлекаем и отображаем информацию из журнала с описанием ошибки. Извлечение информации осуществляется вызовом функции glGetProgramInfoLog. В первом аргументе ей передается дескриптор объекта программы, во втором – размер буфера символов для сохранения записи из журнала, третий – указатель на переменную типа GLsizei , куда будет записано фактическое число символов (без учета заключительного нулевого символа), скопированных в буфер, и четвертый аргумент – указатель на буфер символов для самой записи из журнала. Необходимый объем памяти для буфера можно определить вызовом glGetProgramiv с аргументом GL_INFO_LOG_LENGTH. Строка, возвращаемая в переменной log, уже будет оканчиваться нулевым символом.

Наконец, если компоновка прошла успешно, программа добавляется в конвейер OpenGL вызовом функции glUseProgram, которой в качестве аргумента передается дескриптор объекта программы.

После компиляции фрагментного шейдера из этого рецепта и вершинного шейдера из предыдущего рецепта, компоновки и добавления программы в конвейер OpenGL мы получаем законченный конвейер OpenGL и готовы приступить к отображению. Если попытаться нарисовать треугольник с разными значениями в атрибуте Color, мы получим изображение многоцветного треугольника с вершинами красного, зеленого и синего цветов и с плавными переходами цветов внутри, как показано на рис. 1.3.

И еще…

В одной программе можно создать множество шейдерных программ. Их можно менять в конвейере OpenGL вызовом функции glUseProgram, переключаясь на нужную программу.

Удаление шейдерной программы

Если программа стала не нужна, ее можно удалить из памяти OpenGL вызовом функции glDeleteProgram, передав ей дескриптор объекта программы в единственном аргументе. Это сделает дескриптор программы недействительным и освободит память, занятую программой. Обратите внимание, что если в настоящий момент программа используется, она будет удалена, только когда перестанет использоваться.

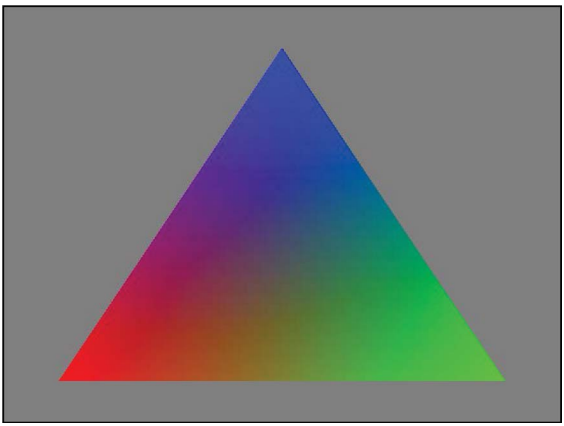


Рис. 1.3 ❖ Многоцветный треугольник с плавными переходами цветов

Удаление программы приведет также к отключению объектов шейдеров, не не удалит их, если прежде не была выполнена операция удаления шейдеров вызовом функции `glDeleteShader`.

См. также

- Рецепт «Компиляция шейдера».

Передача данных в шейдер с использованием вершинных атрибутов и вершинных буферных объектов

Вершинный шейдер вызывается один раз для каждой вершины. Его главная задача – обработать данные, ассоциированные с вершиной, и передать их (и, возможно, другую информацию) на следующий этап конвейера. Чтобы передать вершинному шейдеру что-то для обработки, необходим некоторый механизм передачи исходных данных (для каждой вершины) в шейдер. Обычно такие исходные данные включают (помимо всего прочего) координаты вершины, вектор нормали и координаты текстуры. В прежних версиях OpenGL (до версии 3.0) для передачи каждого элемента информации о вершине имелись отдельные «каналы» – различные функции, такие как `glVertex`, `glTexCoord` и `glNormal` (или внутри клиентских массивов вершин, с использованием `glVertexPointer`, `glTexCoordPointer` или `glNormalPointer`). Шейдер мог обращаться к значениям, переданным таким способом, посредством встроенных переменных, таких как `gl_Vertex` и `gl_Normal`. Но в версии OpenGL 3.0 эта функциональность была объявлена устаревшей и позднее удалена из библиотеки. Теперь информация о вершинах должна передаваться

через *универсальные вершинные атрибуты*, обычно в комплексе с (вершинными) *буферными объектами*. Программист сейчас может определить произвольный набор атрибутов для каждой вершины с целью передачи информации в вершинный шейдер. Например, чтобы реализовать рельефное текстурирование методом наложения нормалей (normal mapping), программист может решить передавать для каждой вершины не только ее координаты, но также вектор нормали и вектор касательной. В OpenGL 4 это легко реализовать, определив комплект входных атрибутов. Такой подход определения информации для каждой вершины не только обеспечивает большую гибкость, но и может потребовать времени на освоение для тех, кто привык пользоваться устаревшими приемами.

В вершинном шейдере определение входных атрибутов выполняется с использованием квалификатора `in`. Например, определить входной атрибут с трехкомпонентным вектором `VertexColor` можно следующим образом:

```
in vec3 VertexColor;
```

Разумеется, данные для этого атрибута должны поставляться программой OpenGL. Для этого используются вершинные буферные объекты (vertex buffer objects). Буферный объект содержит значения для входного атрибута. Основная программа связывает буферный объект с входным атрибутом и определяет, как «шагать» по данным. Затем во время отображения OpenGL будет извлекать из буфера данные для атрибута перед каждым вызовом вершинного шейдера.

В данном рецепте рисуется единственный треугольник. В число необходимых нам атрибутов входят координаты вершины и цвет. Мы будем использовать фрагментный шейдер для интерполяции цветов вершин, чтобы получился треугольник, как показано на рис. 1.4. Вершины треугольника имеют красный, зеленый и синий цвета, а тело раскрашивается путем смешивания этих цветов. Возможно, на бумаге цвета не видны (в черно-белой книге), но по градациям серого цвета можно догадаться о смешивании цветов.

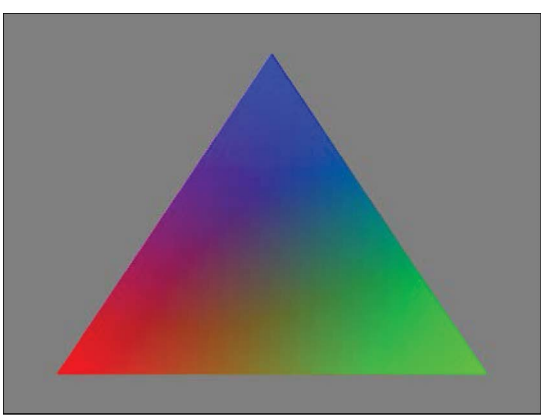
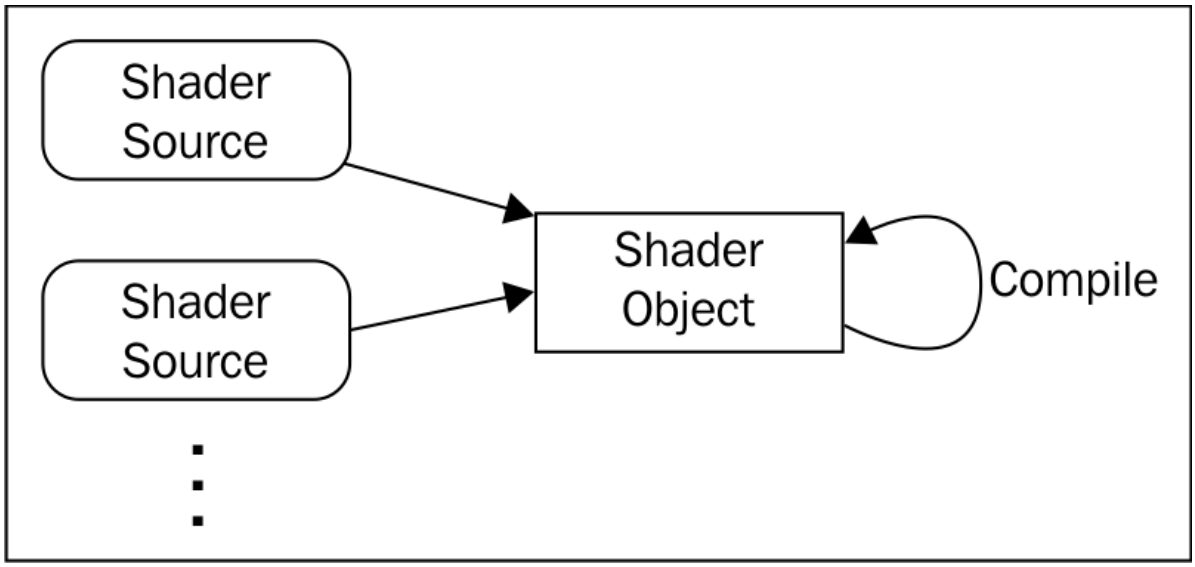
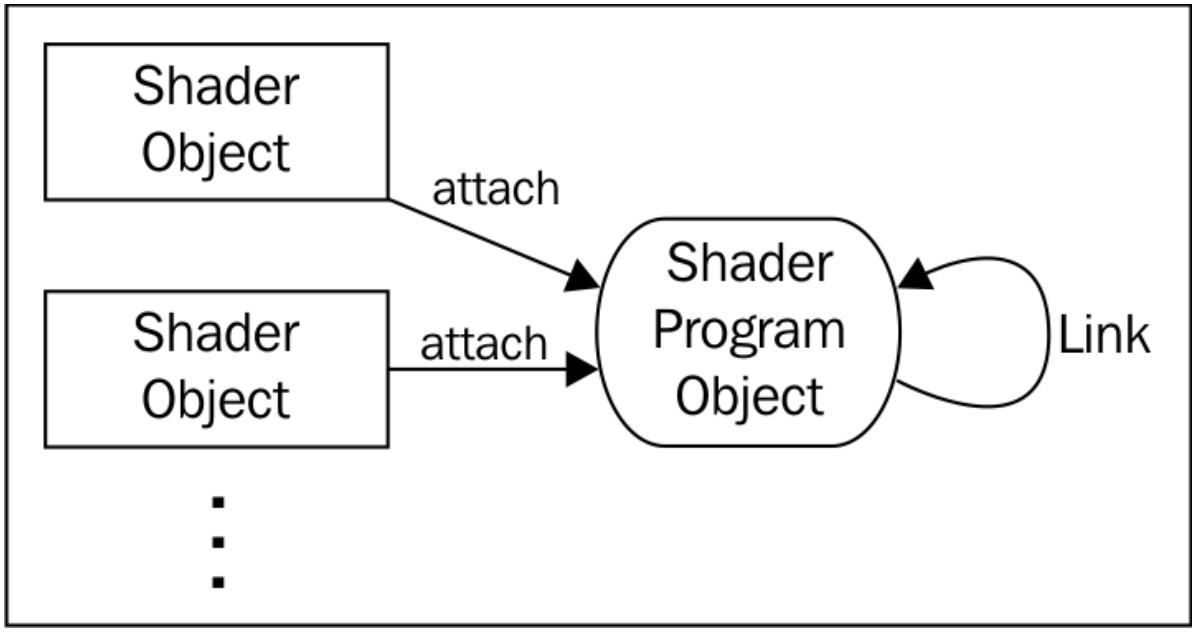
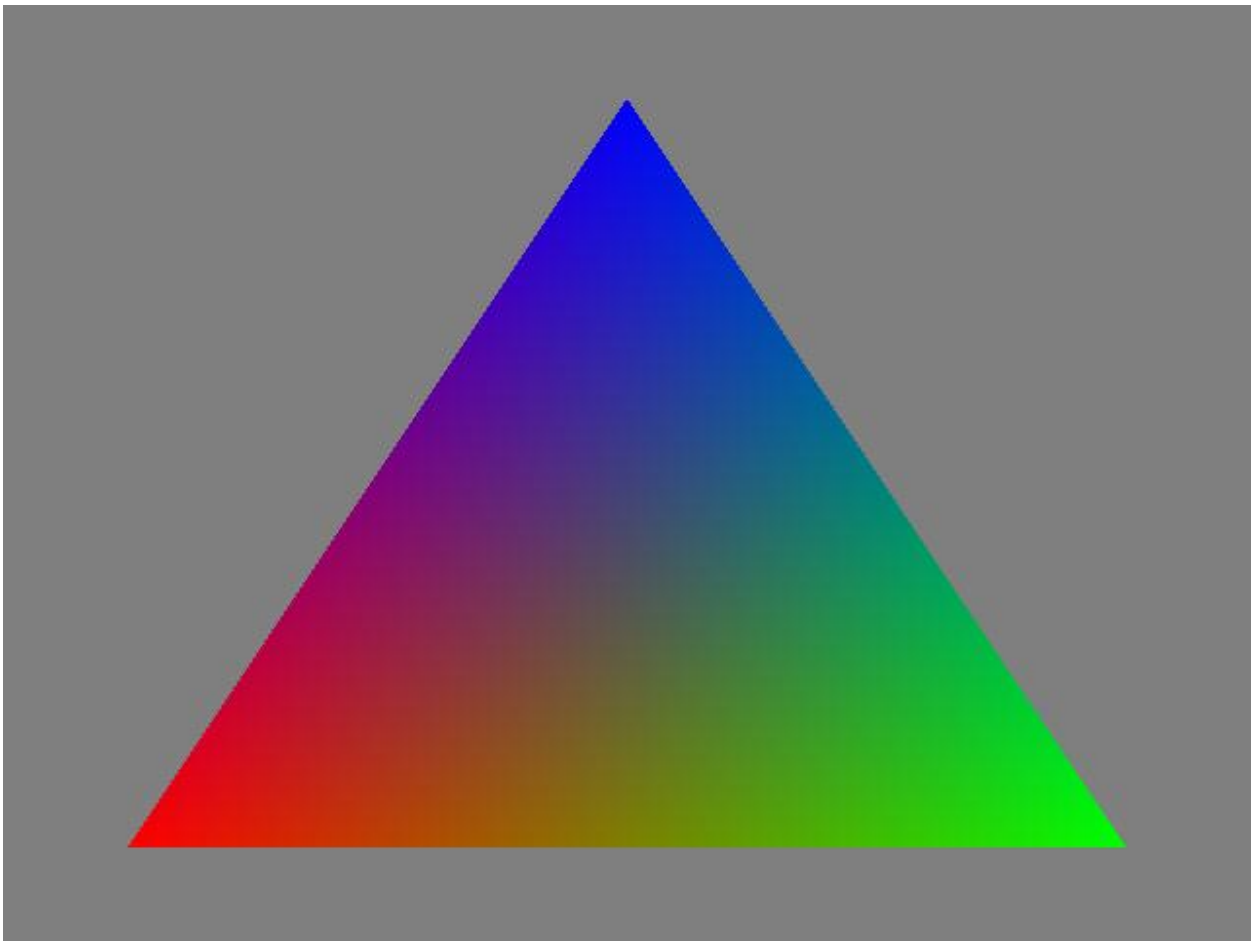
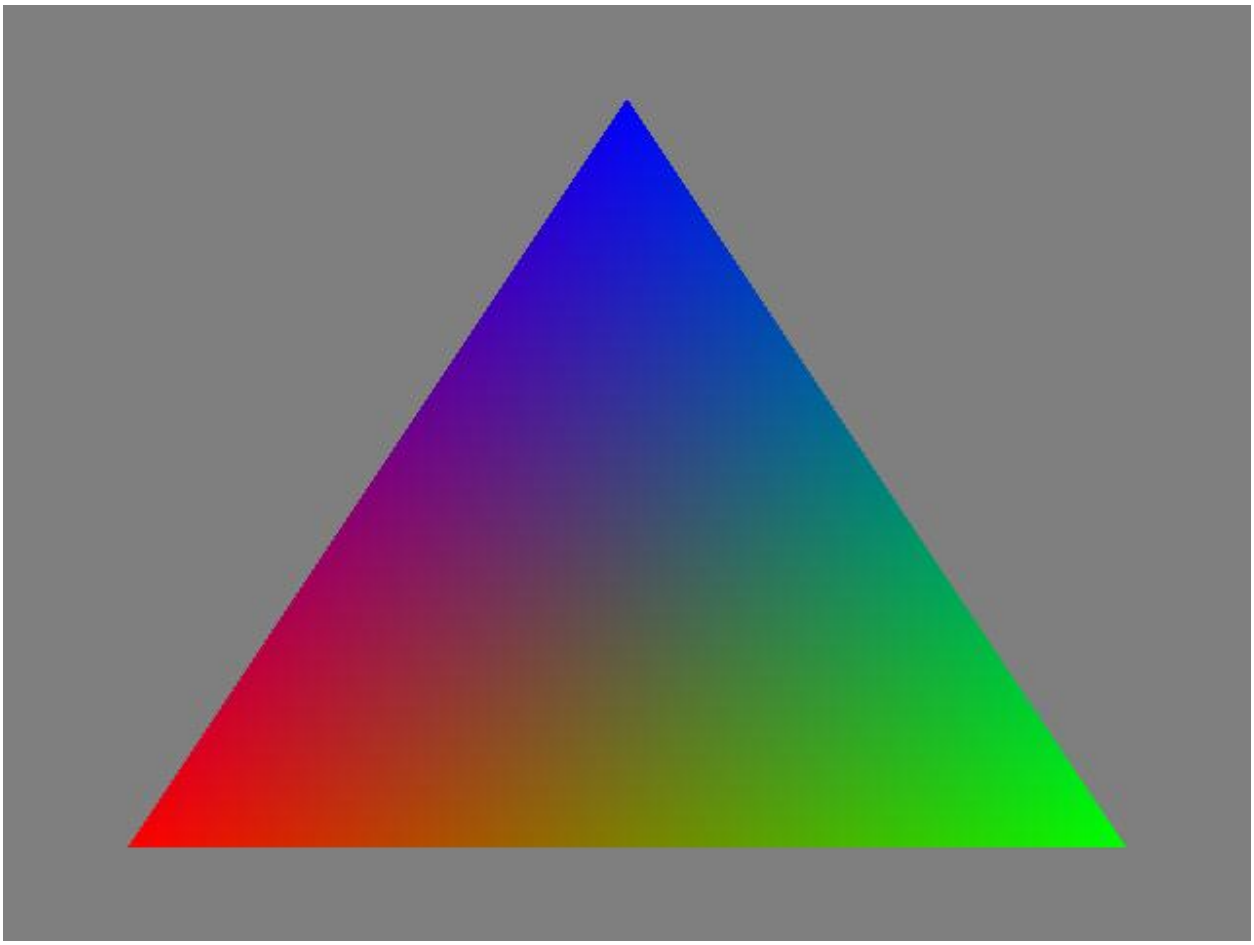


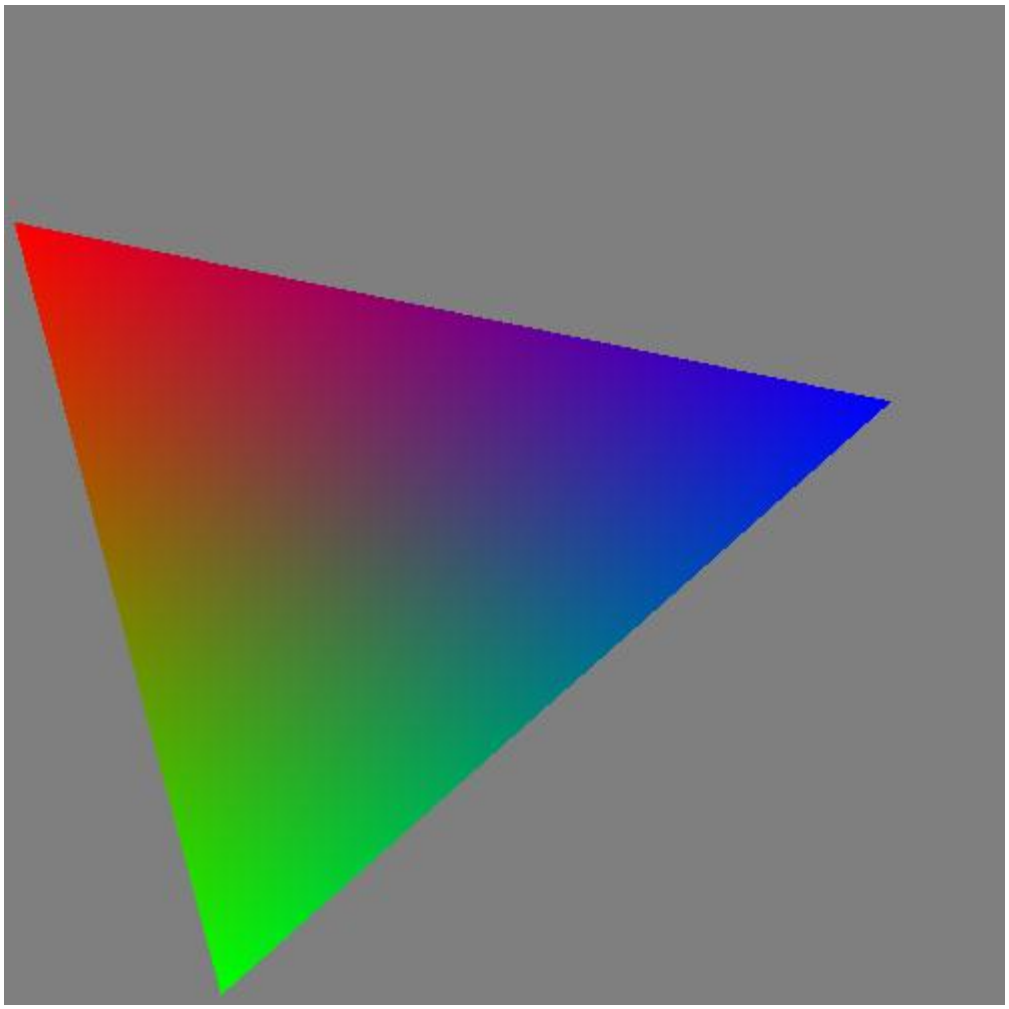
Рис. 1.4 ❖ Результат работы шейдеров в данном рецепте

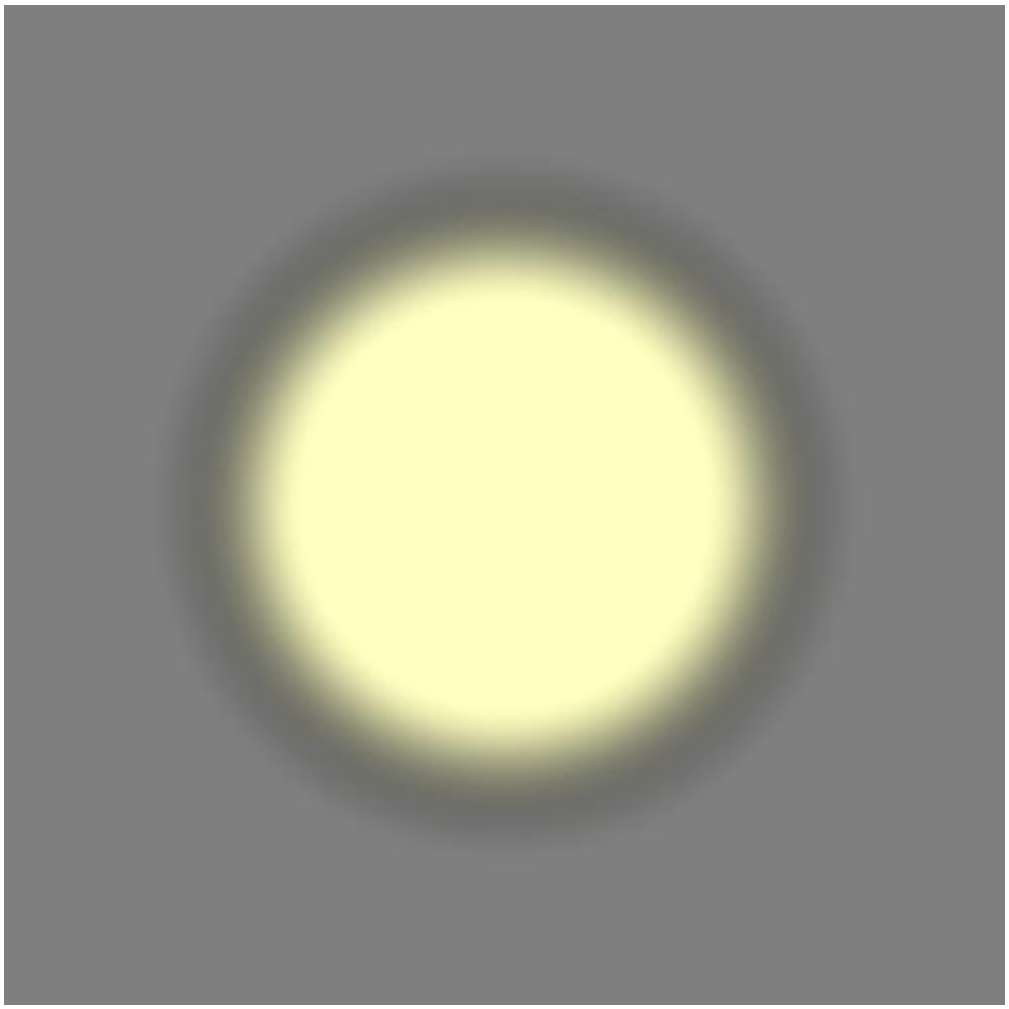

Shader
Source
Shader
Source
Shader
Object
Compile

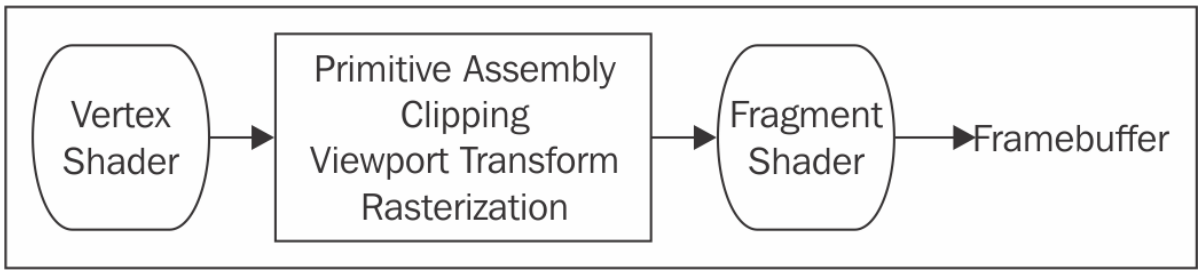

Shader
Object
attach
Shader
Object
attach
Shader
Program
Object
Link

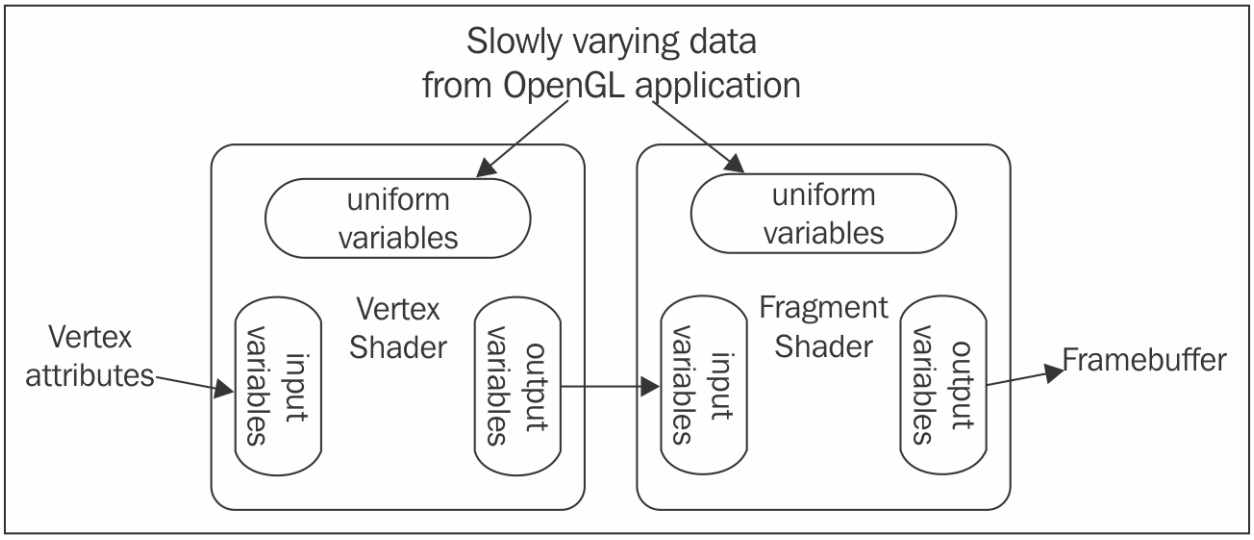






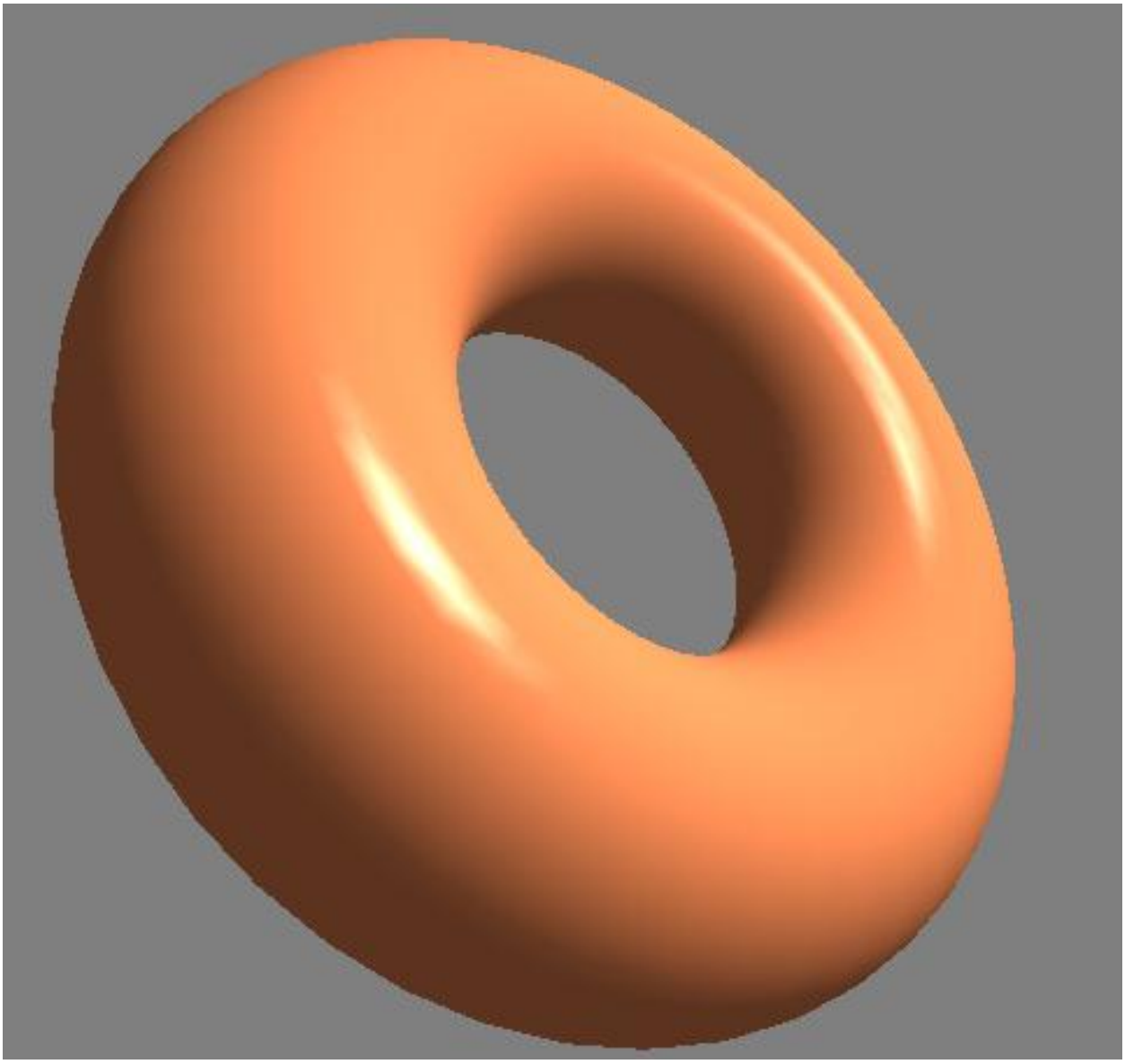



Vertex Shader
Primitive Assembly
Clipping
Viewport Transform
Rasterization
Fragment Shader
Framebuffer



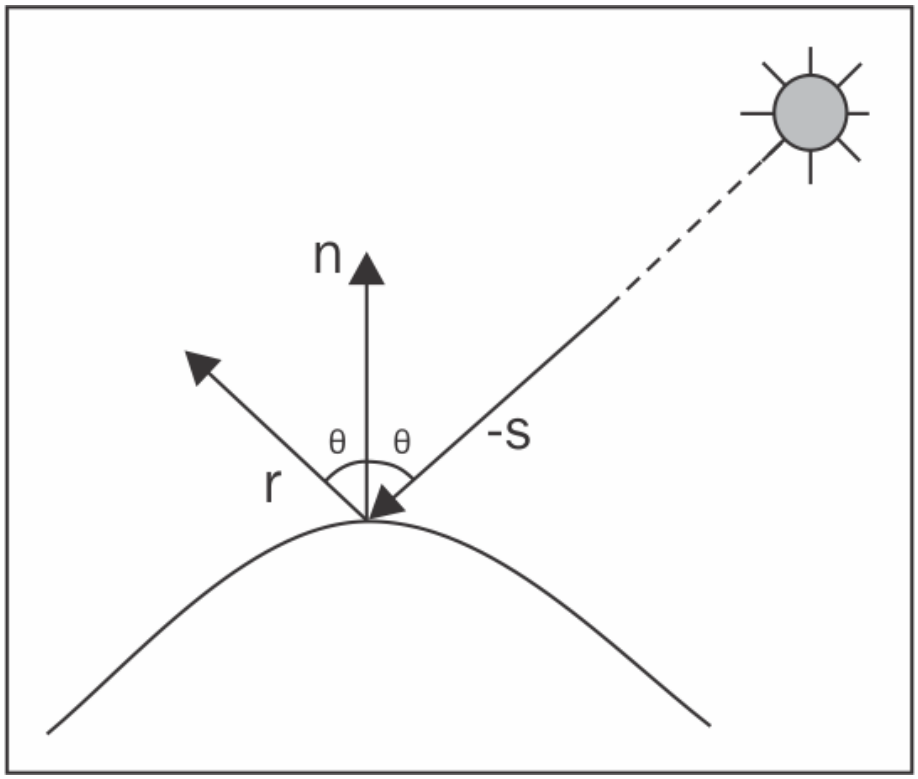
$$L_d \mathbf{s} \cdot \mathbf{n}$$

$$L = K_d L_d \mathbf{s} \cdot \mathbf{n}$$

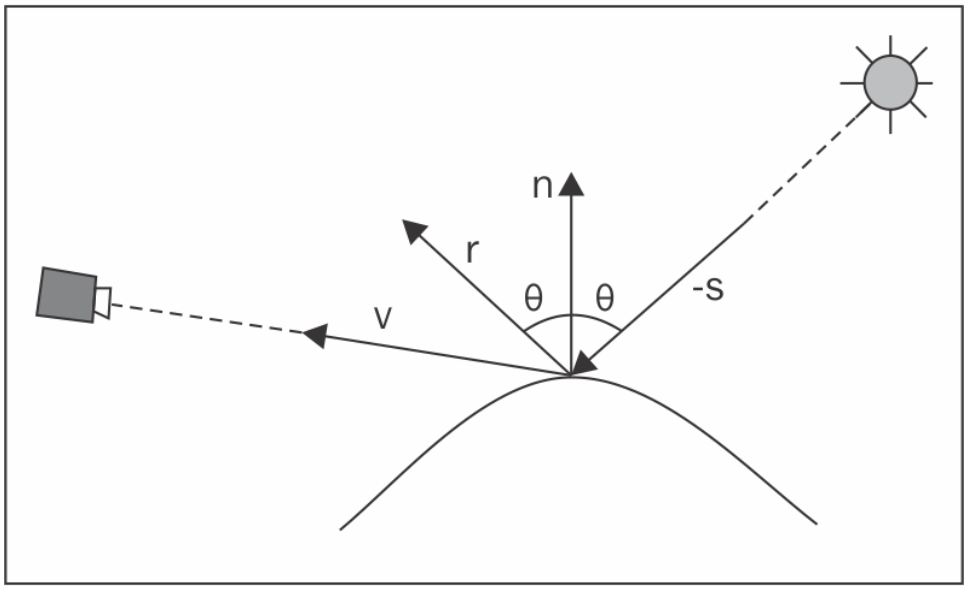


$$I_a = L_a K_a$$

$$I_d = L_d K_d (\mathbf{s} \cdot \mathbf{n})$$

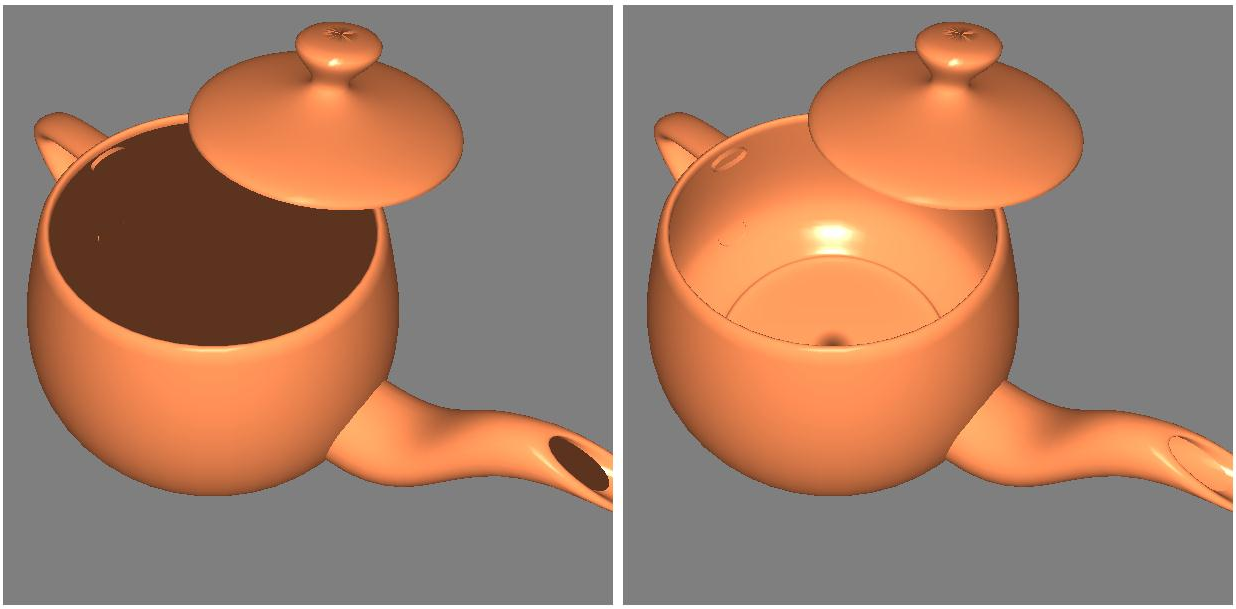


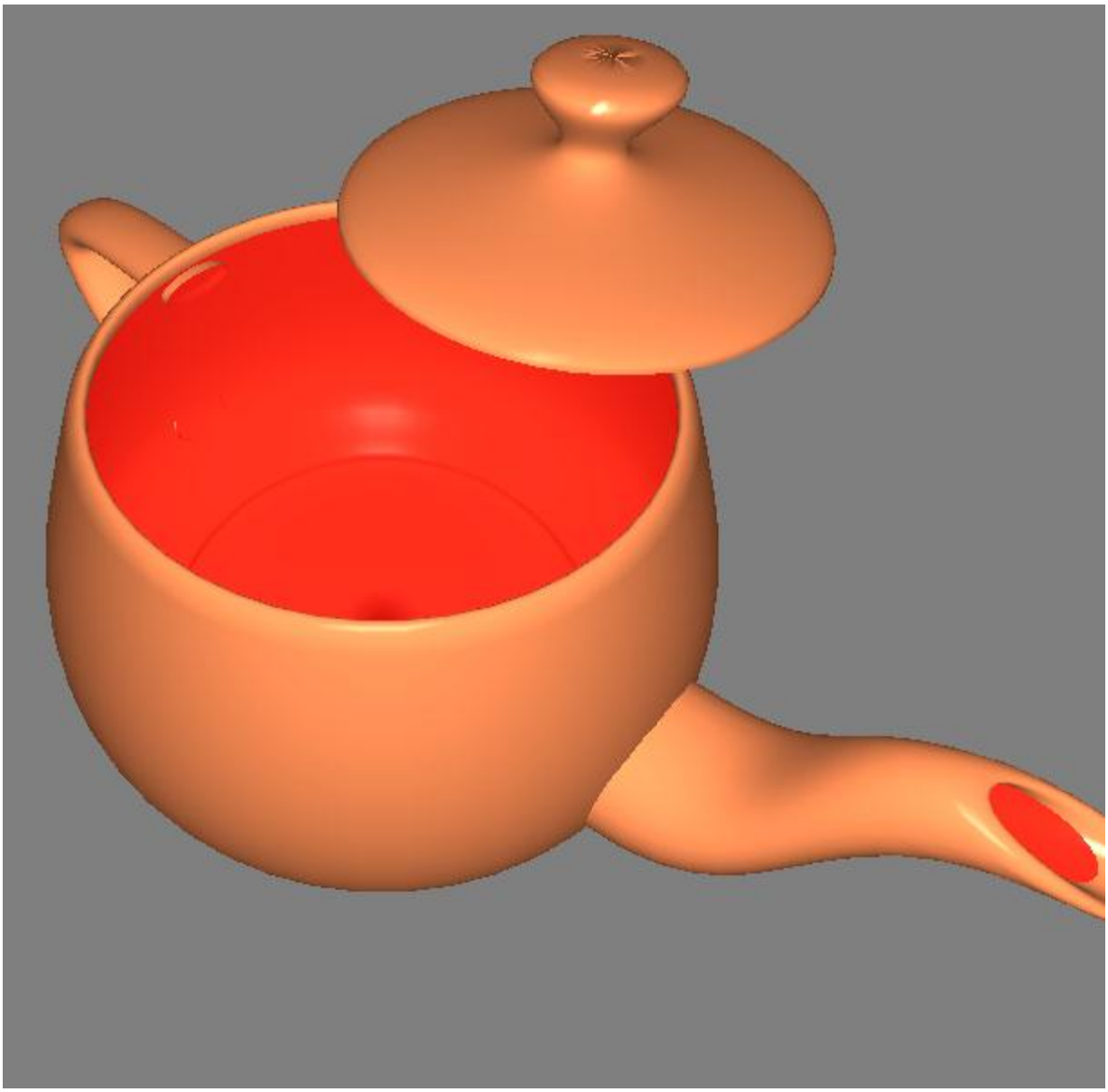
$$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s} \cdot \mathbf{n})\mathbf{n}$$

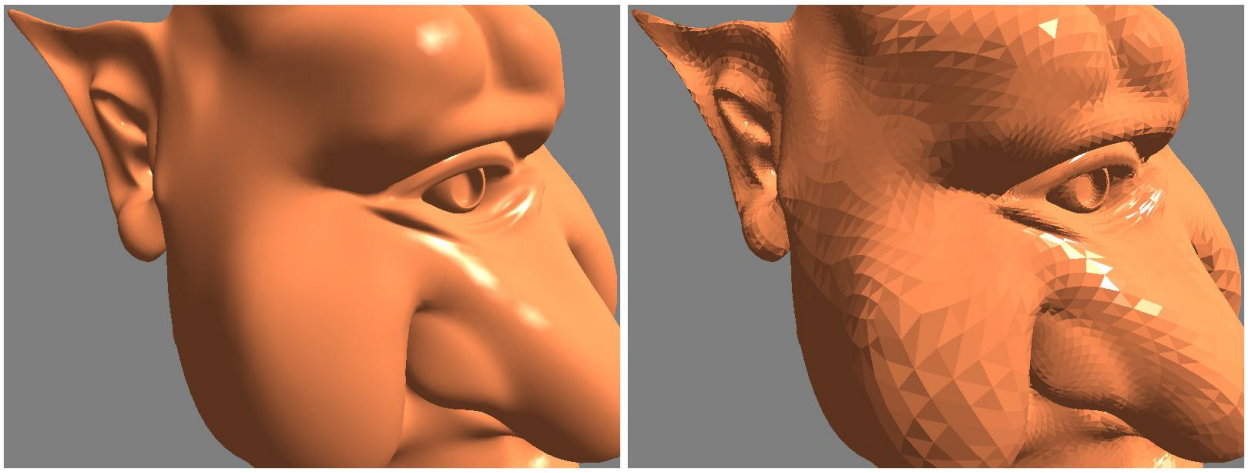


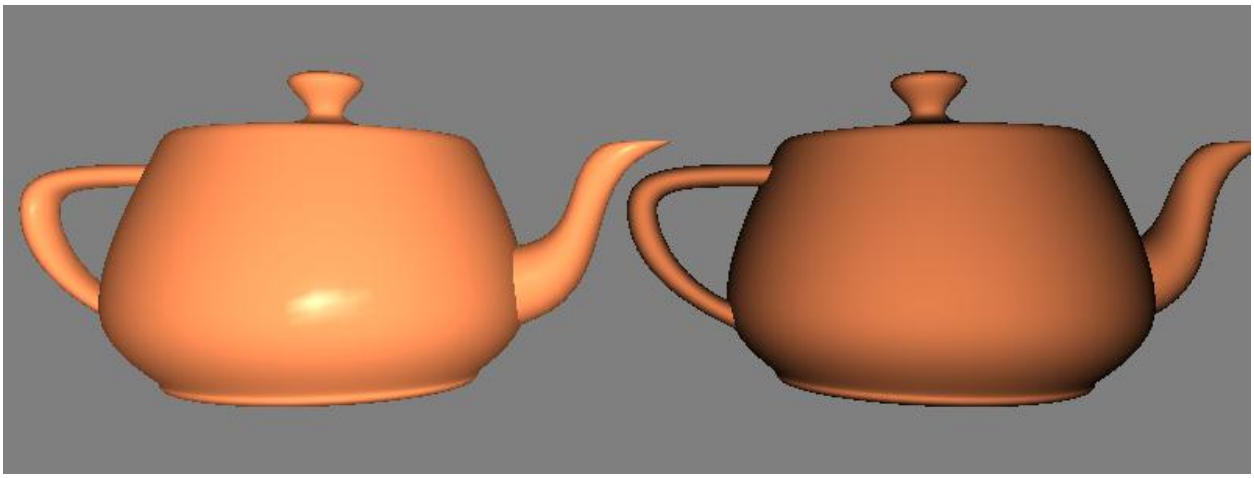
$$I_s = L_s K_s (\mathbf{r} \cdot \mathbf{v})^f$$

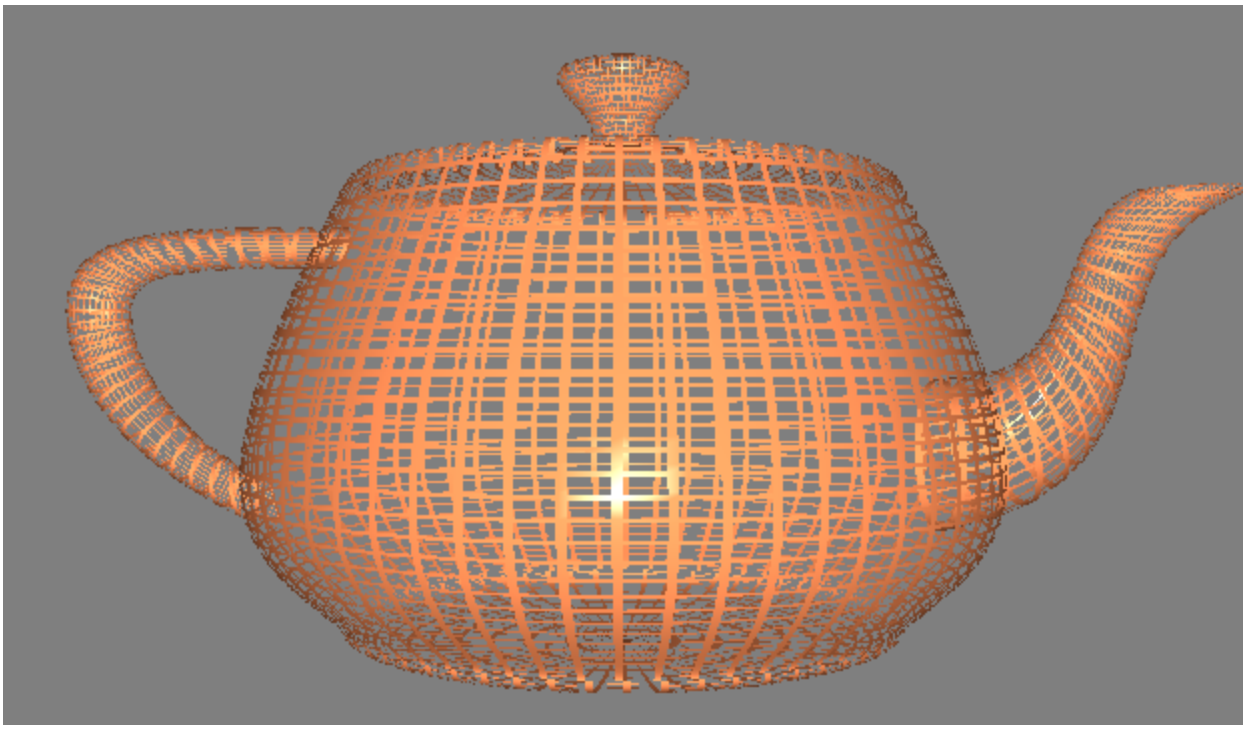
$$\begin{aligned} I &= I_a + I_d + I_s \\ &= L_a K_a + L_d K_d (\mathbf{s} \cdot \mathbf{n}) + L_s K_s (\mathbf{r} \cdot \mathbf{v})^f \end{aligned}$$

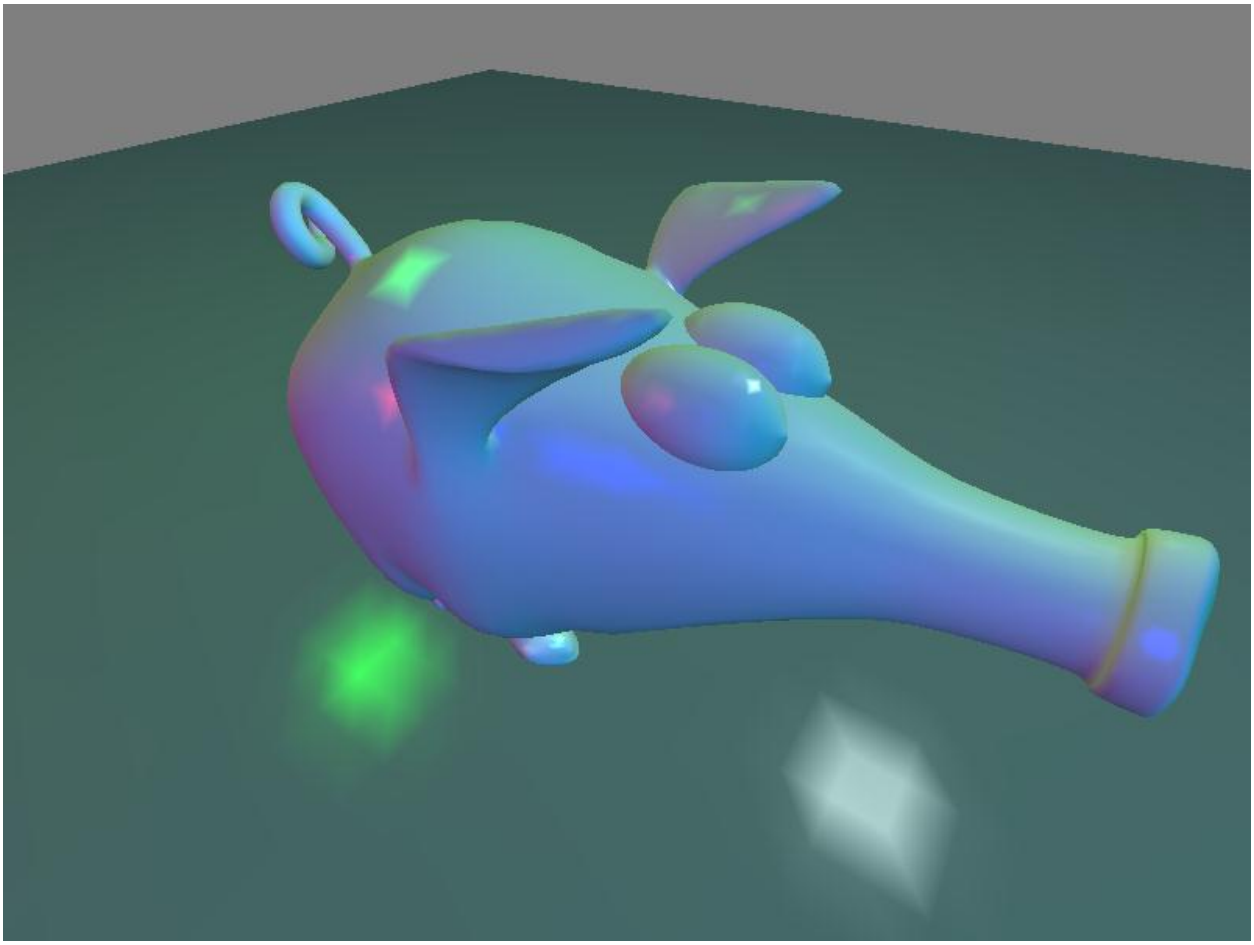


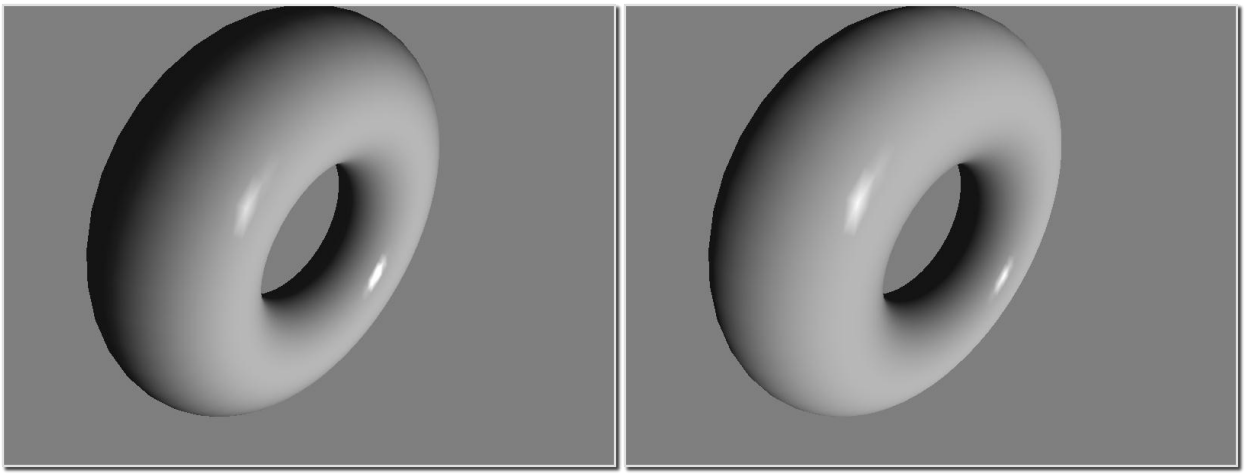


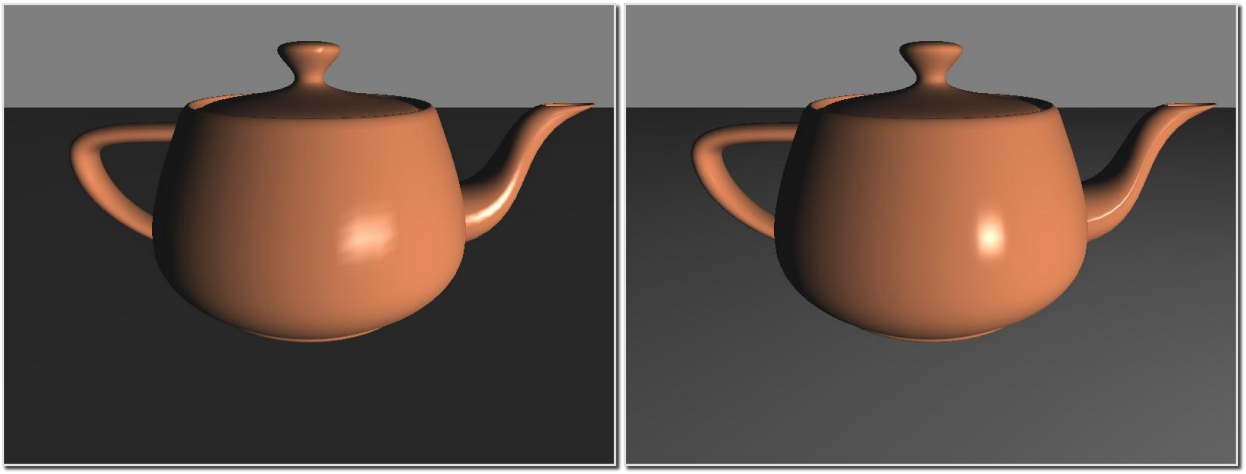








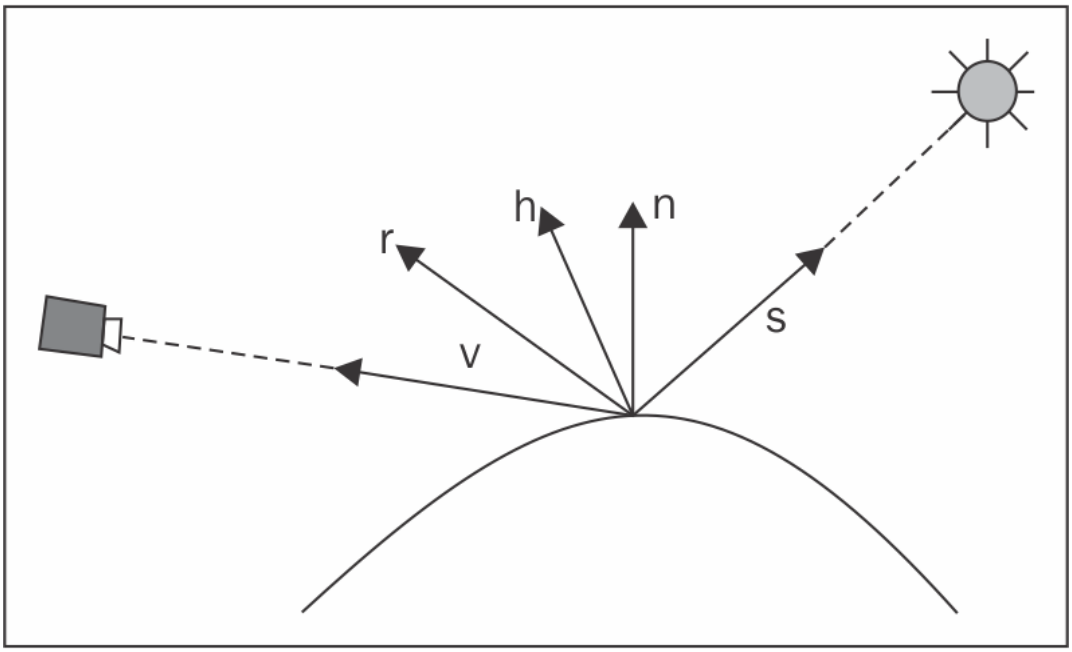




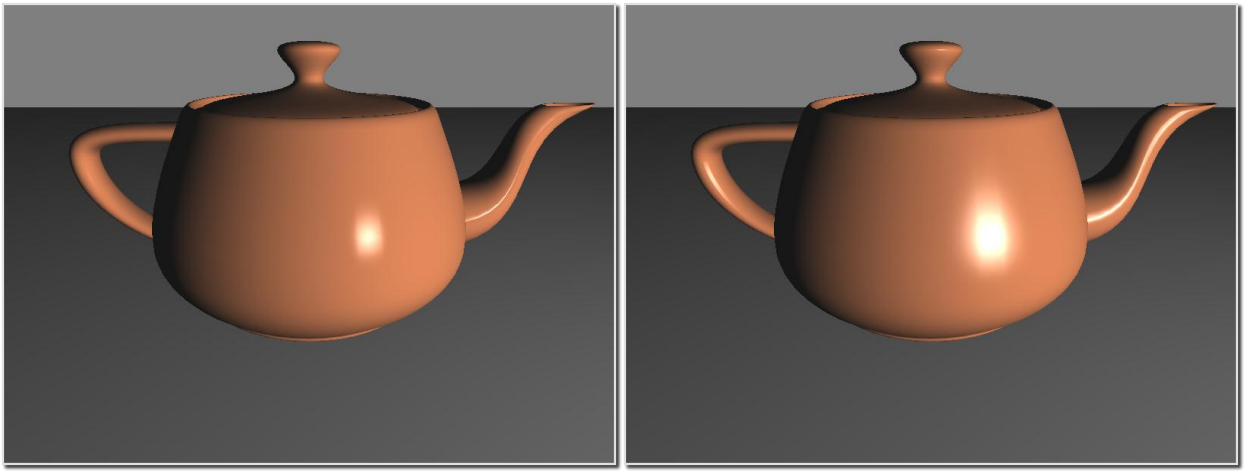
$$I_s = L_s K_s (\mathbf{r} \cdot \mathbf{v})^f$$

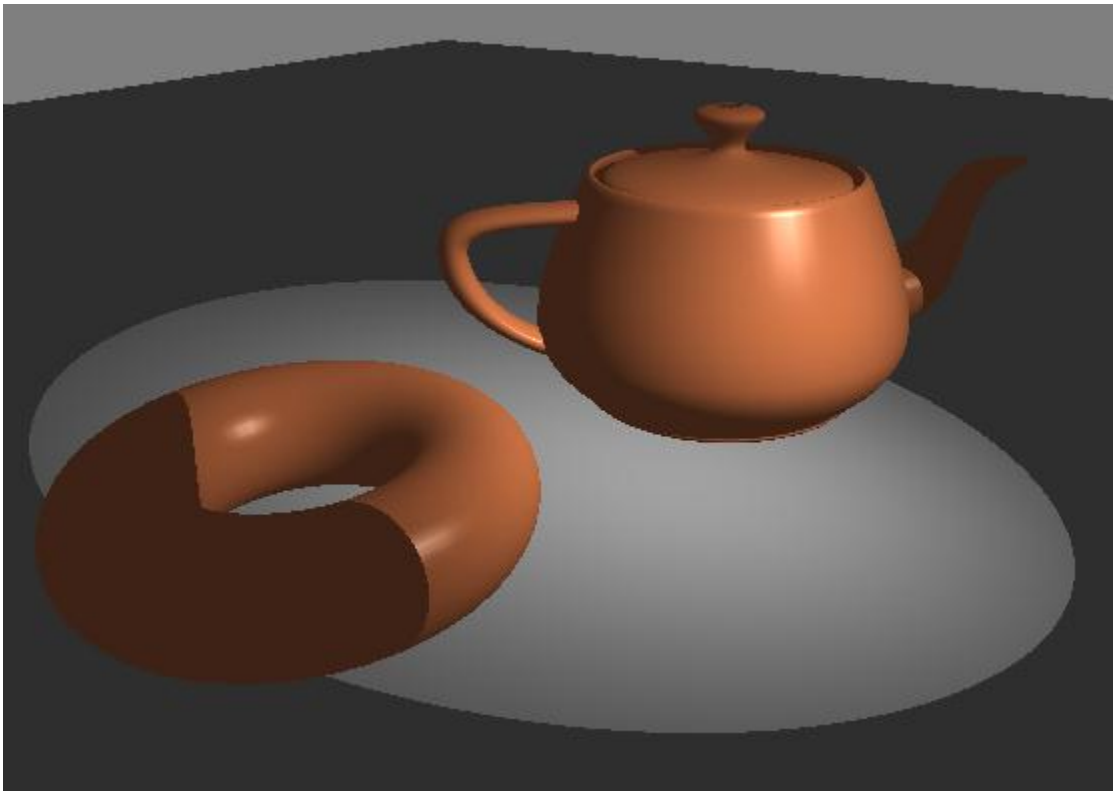
$$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s} \cdot \mathbf{n})\mathbf{n}$$

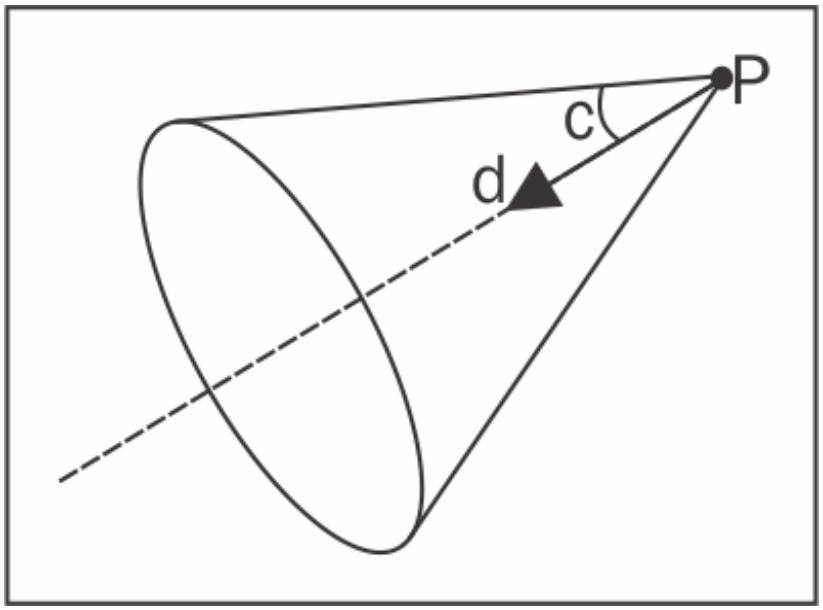
$$\mathbf{h} = \mathbf{v} + \mathbf{s}$$

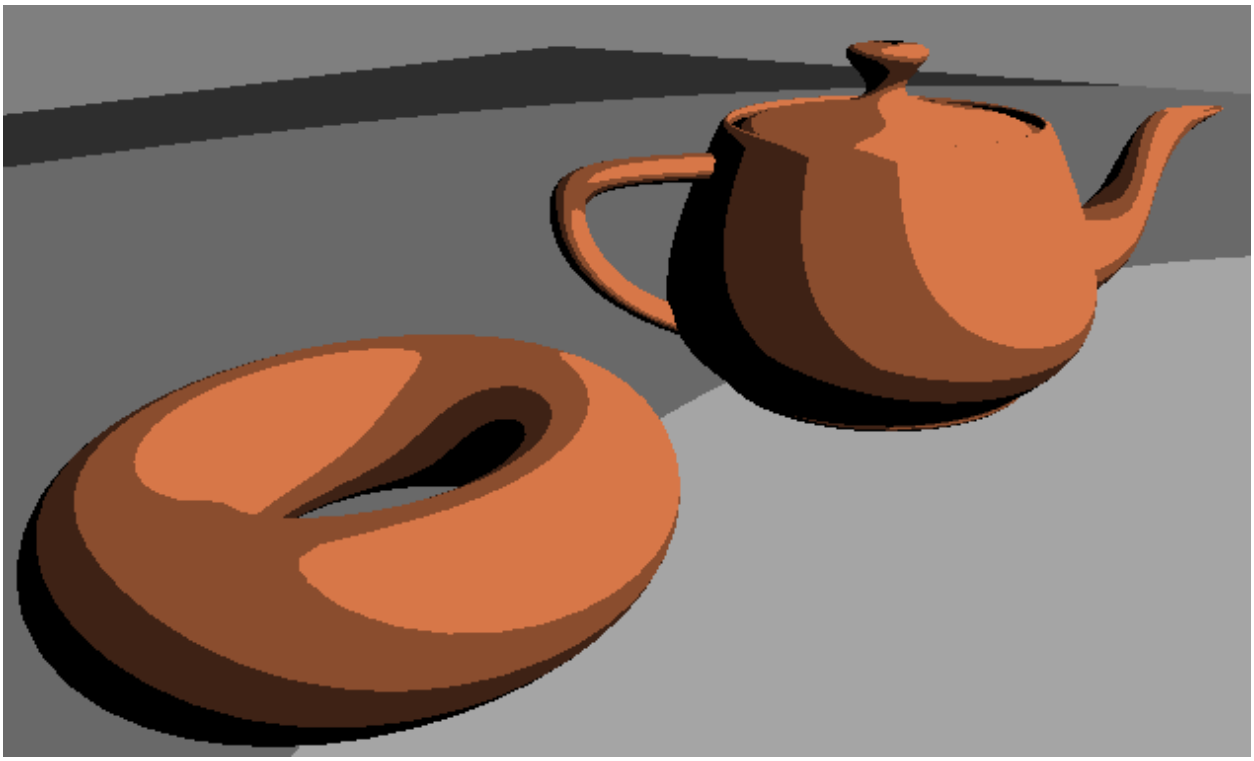


$$I_s = L_s K_s (\mathbf{h} \cdot \mathbf{n})^f$$





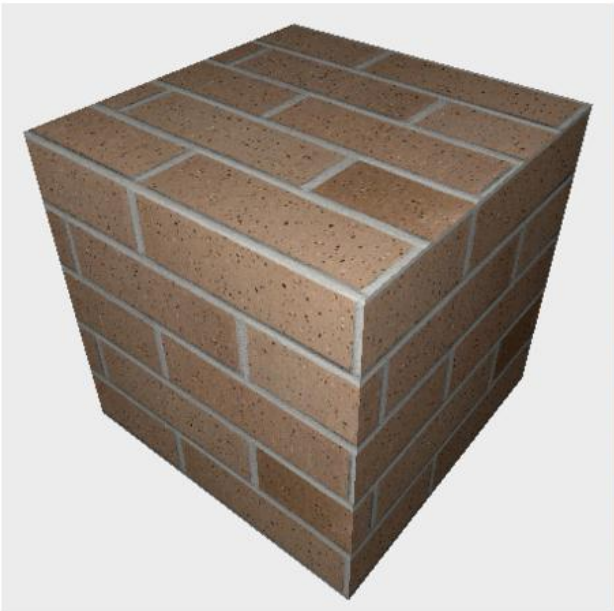

P
c
d

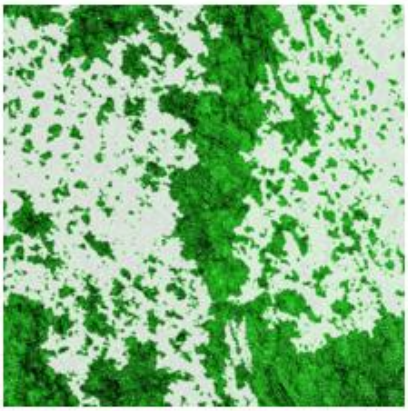


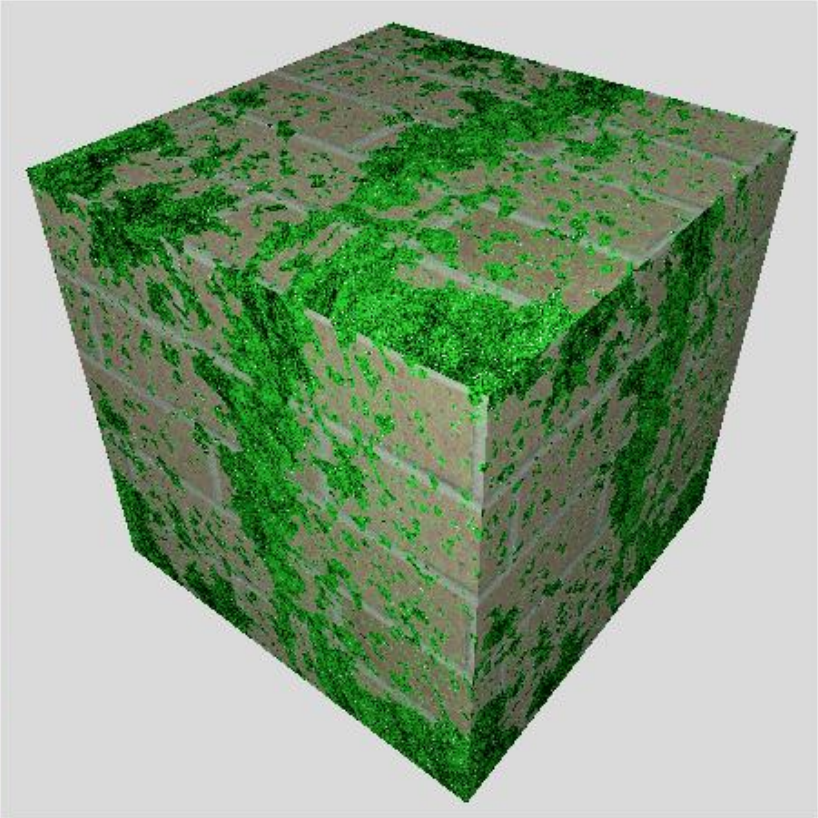
$$f = \frac{d_{\max} - |z|}{d_{\max} - d_{\min}}$$

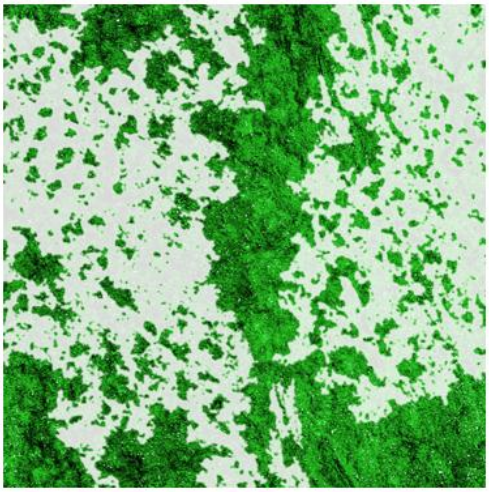
$$f = e^{-d|z|}$$

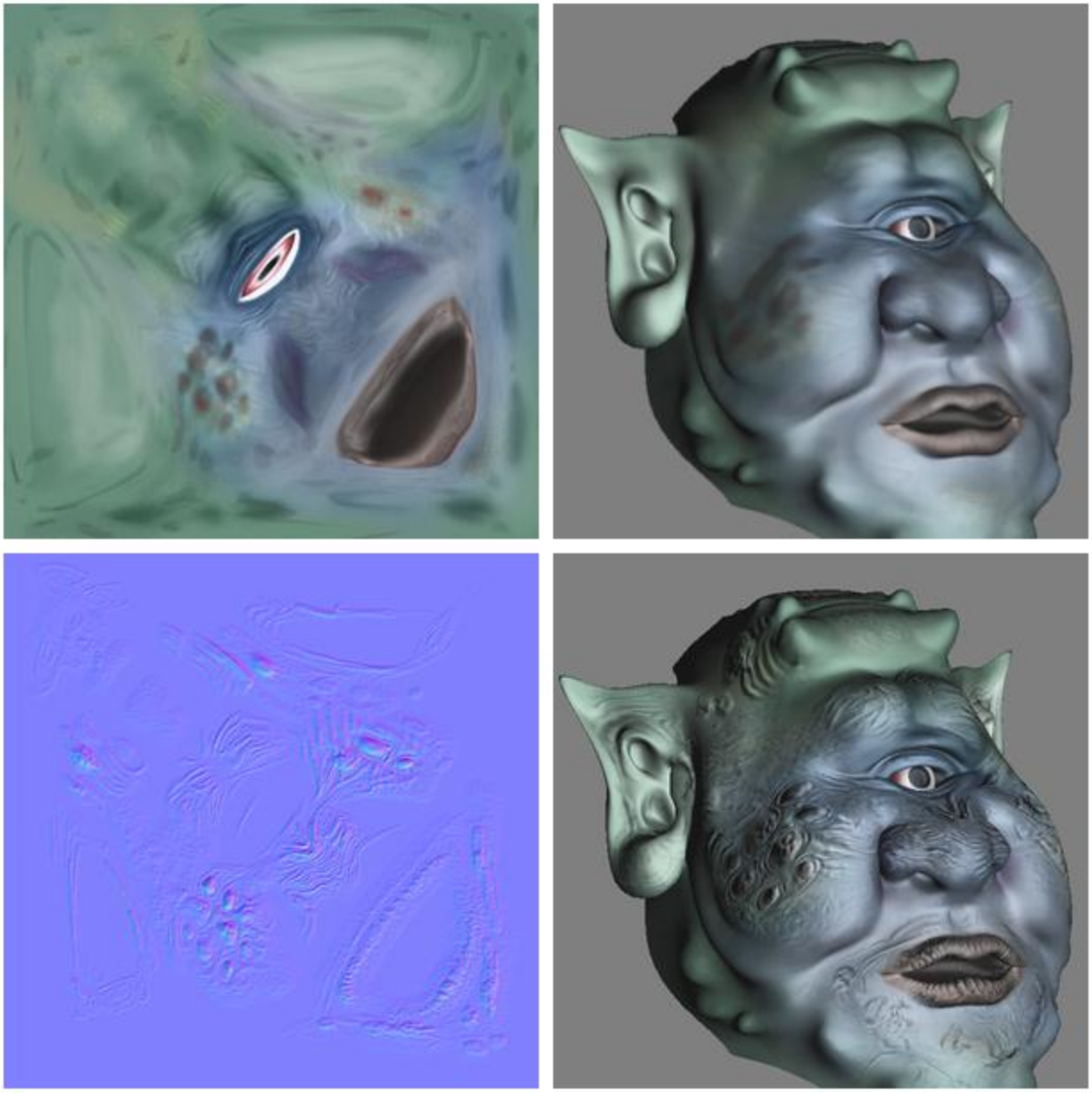
$$f = e^{-(dz)^2}$$

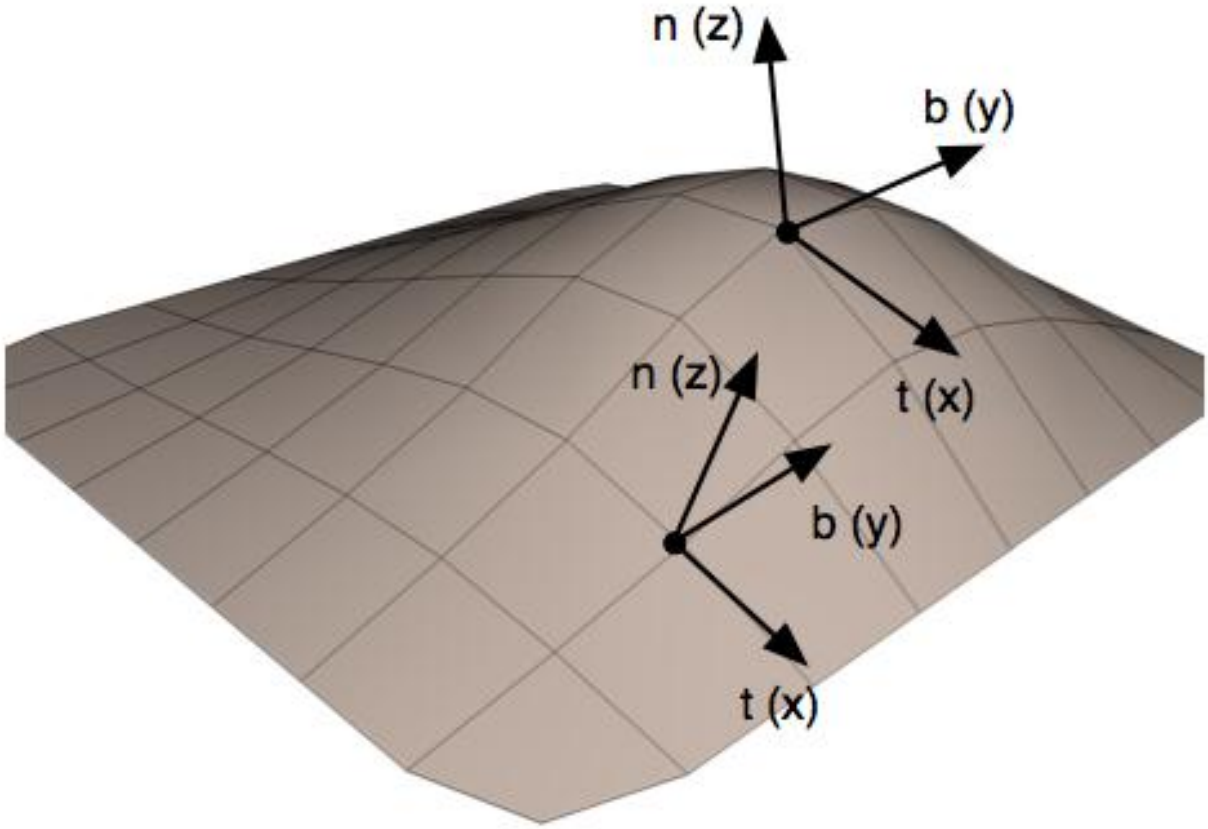




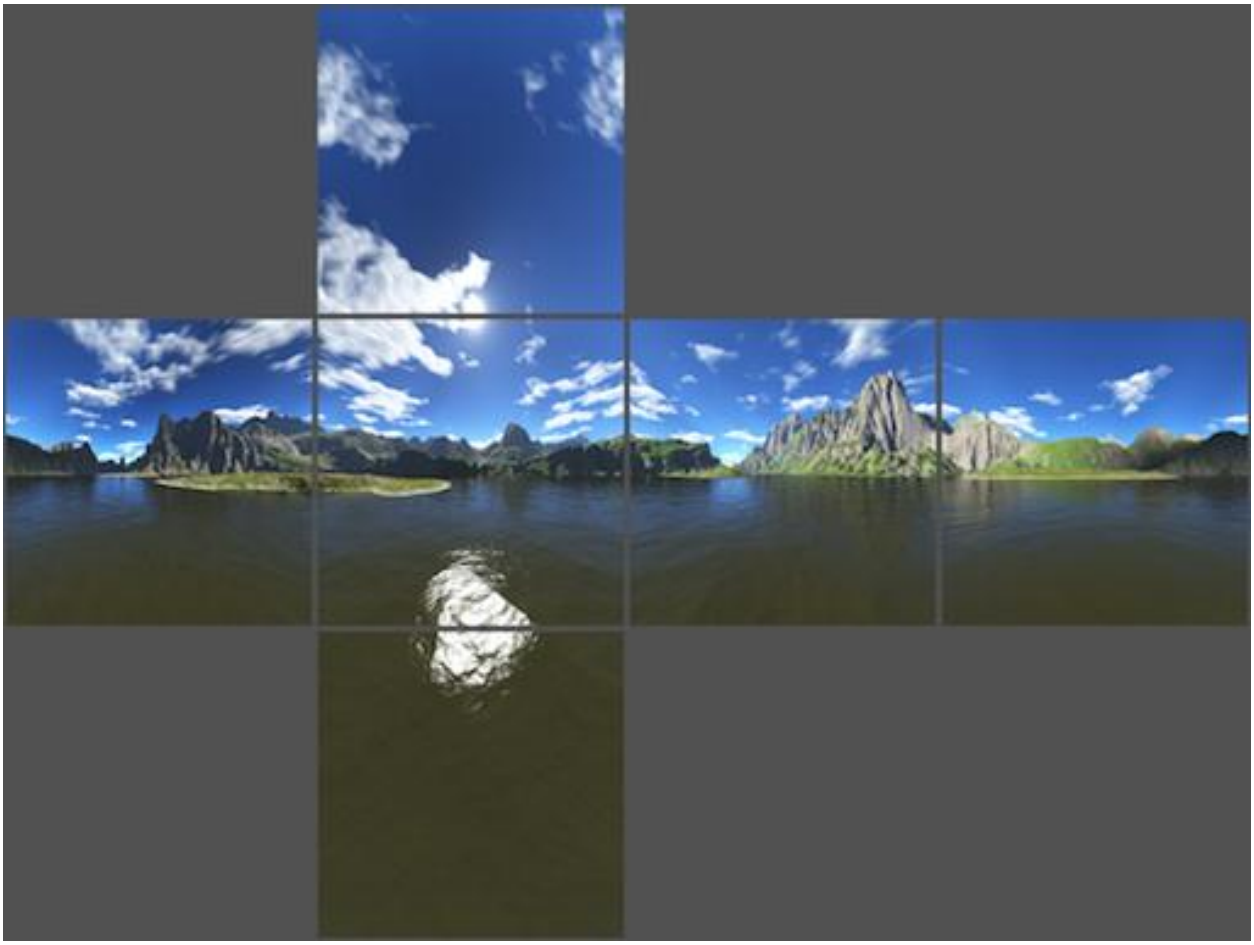


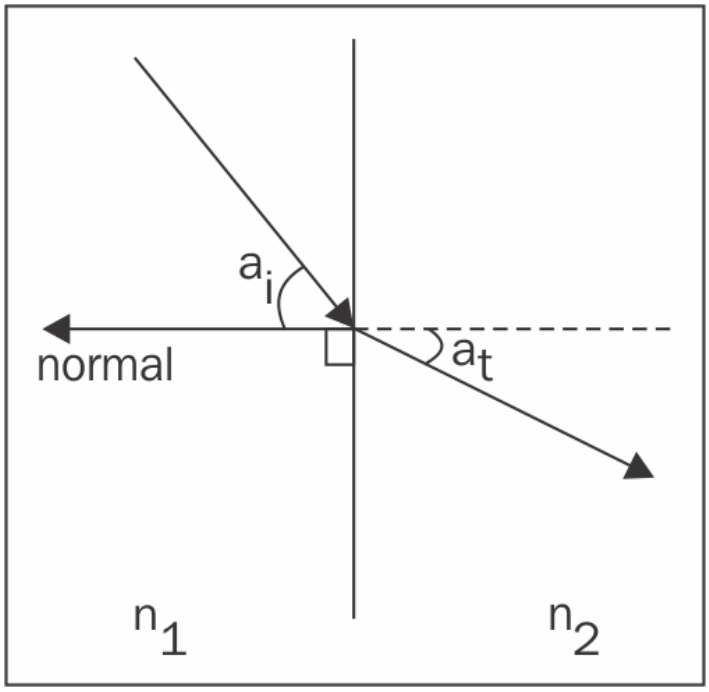




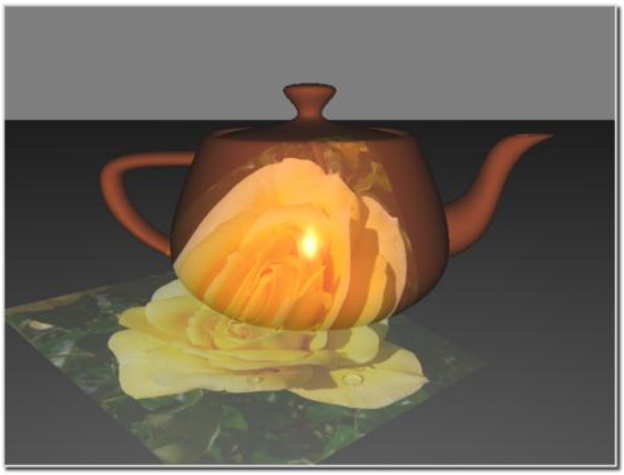


$$\begin{bmatrix} S_x \\ S_y \\ S_z \end{bmatrix} = \begin{bmatrix} t_x & t_y & t_z \\ b_x & b_y & b_z \\ n_x & n_y & n_z \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$$

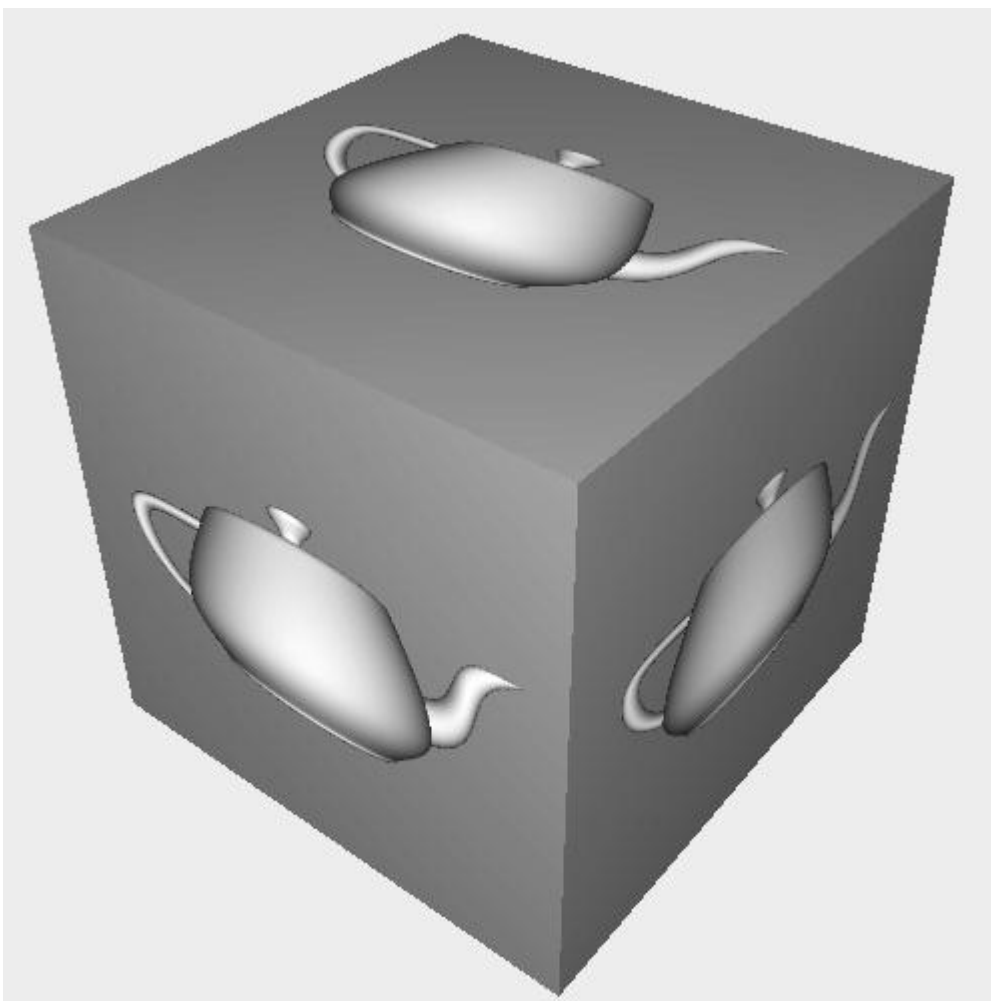


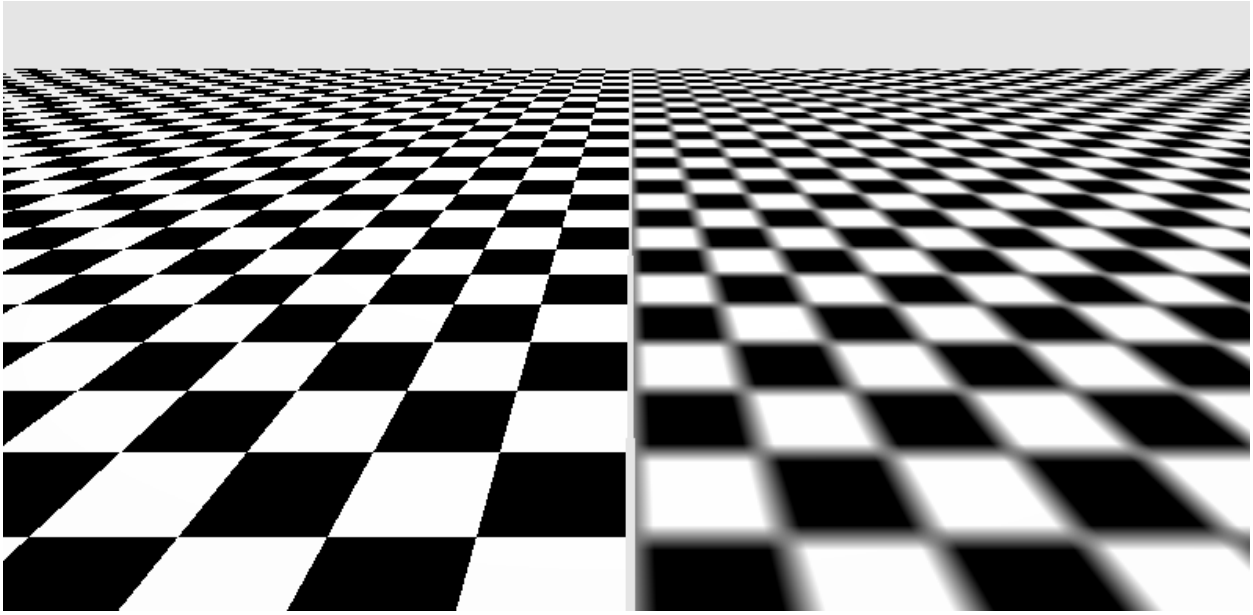


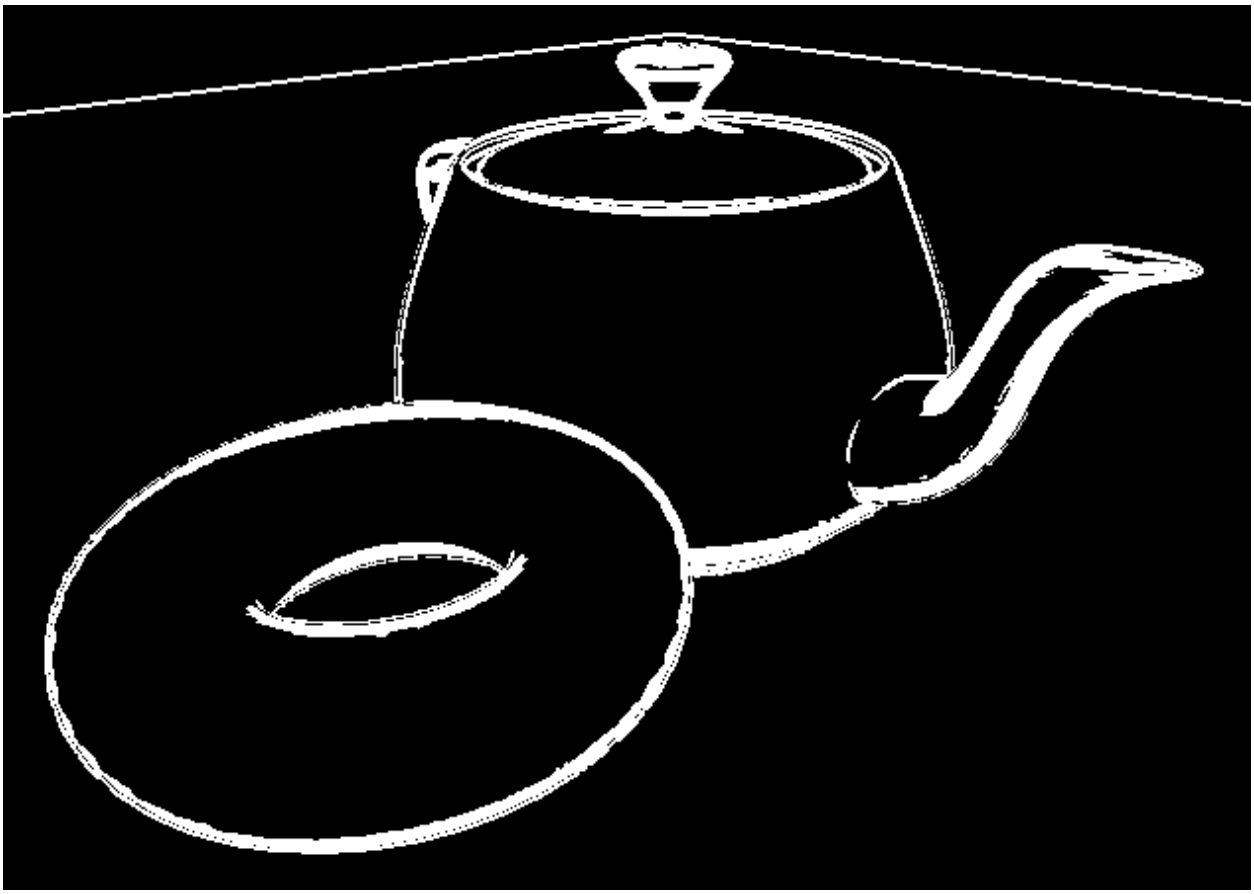
$$\frac{\sin a_i}{\sin a_t} = \frac{n_2}{n_1}$$



$$\mathbf{M} = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{PV}$$





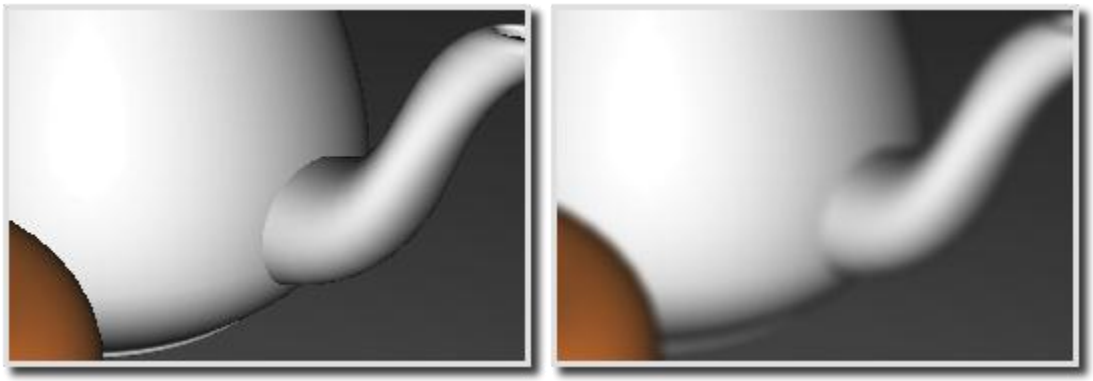


10	11	12	13	14
1 17	0 18	1 19	20	21
0 24	2 25	0 26	27	28
1 31	0 32	1 33	34	35
38	39	40	41	42

$$\mathbf{S}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \mathbf{S}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

$$g = \sqrt{s_x^2 + s_y^2}$$

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$



$$G(x,y) = G(x)G(y)$$

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

$$C_{lm} \leftarrow \sum_{i=-4}^{4} \sum_{j=-4}^{4} G(i,j) C_{l+i\ m+j}$$

$$C_{lm} \leftarrow \sum_{i=-4}^{4} G(i) \sum_{j=-4}^{4} G(j) C_{l+i\ m+j}$$

$$C_{lm} \leftarrow \sum_{i=-4}^{4} \frac{G(i)}{k} \sum_{j=-4}^{4} \frac{G(j)}{k} C_{l+i\ m+j}$$

$$k = \sum_{i=-4}^{4} G(i)$$

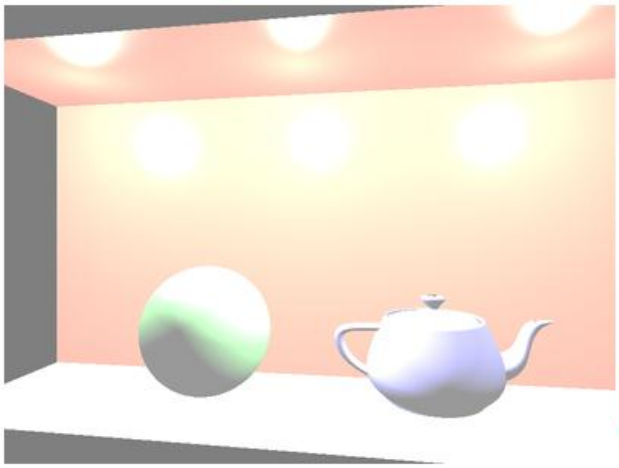
$$\bar{L}_w = \exp\left(\frac{1}{N}\sum_{x,y}\ln(0.0001 + L_w(x,y))\right)$$

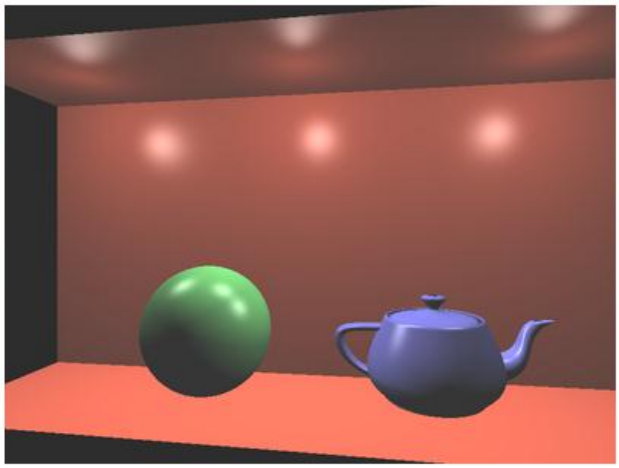
$$L(x,y) = \frac{a}{\bar{L}_w} L_w(x,y)$$

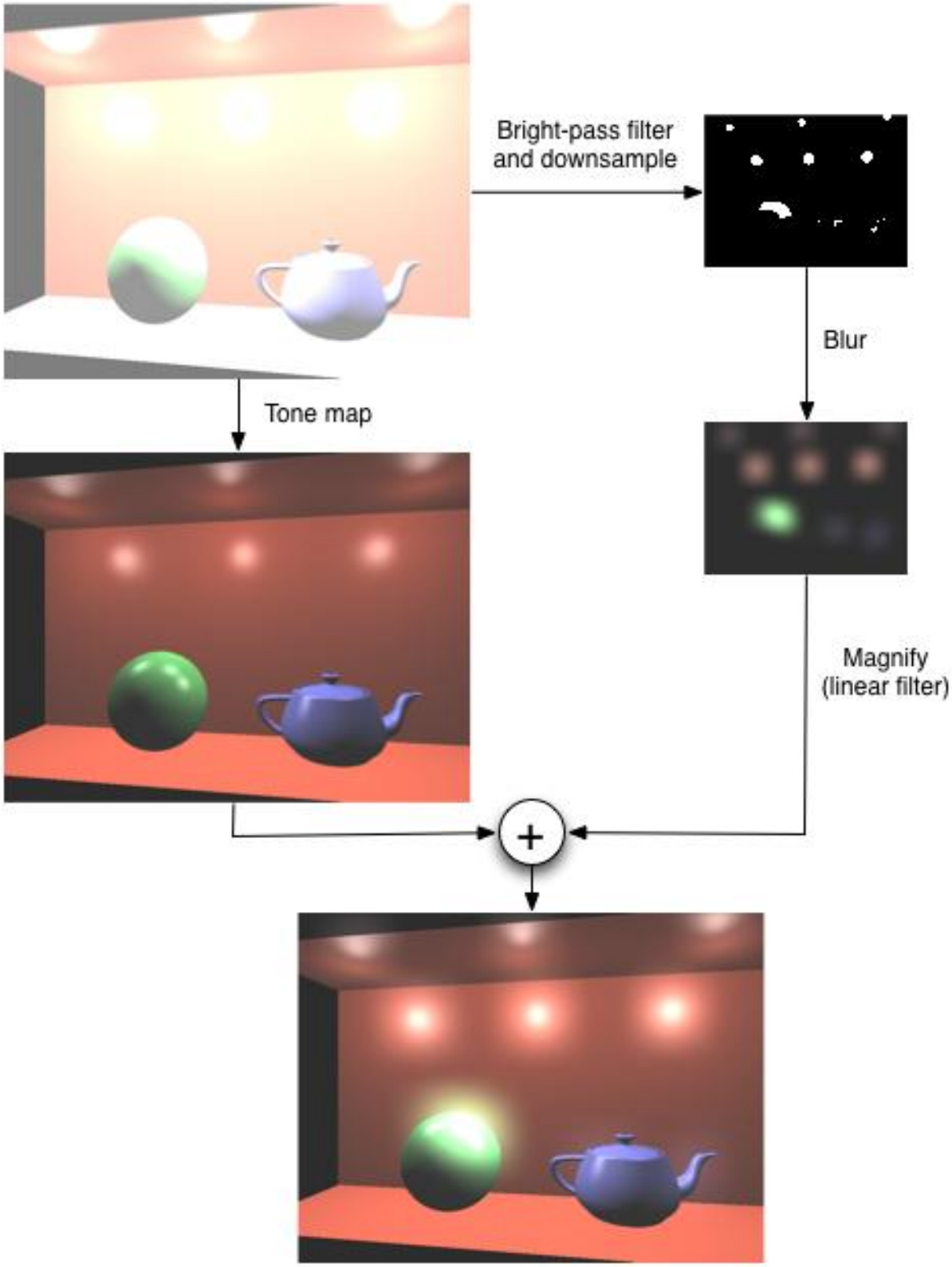
$$L_d(x,y) = \frac{L(x,y)\left(1 + \frac{L(x,y)}{L^2_{\text{white}}}\right)}{1 + L(x,y)}$$

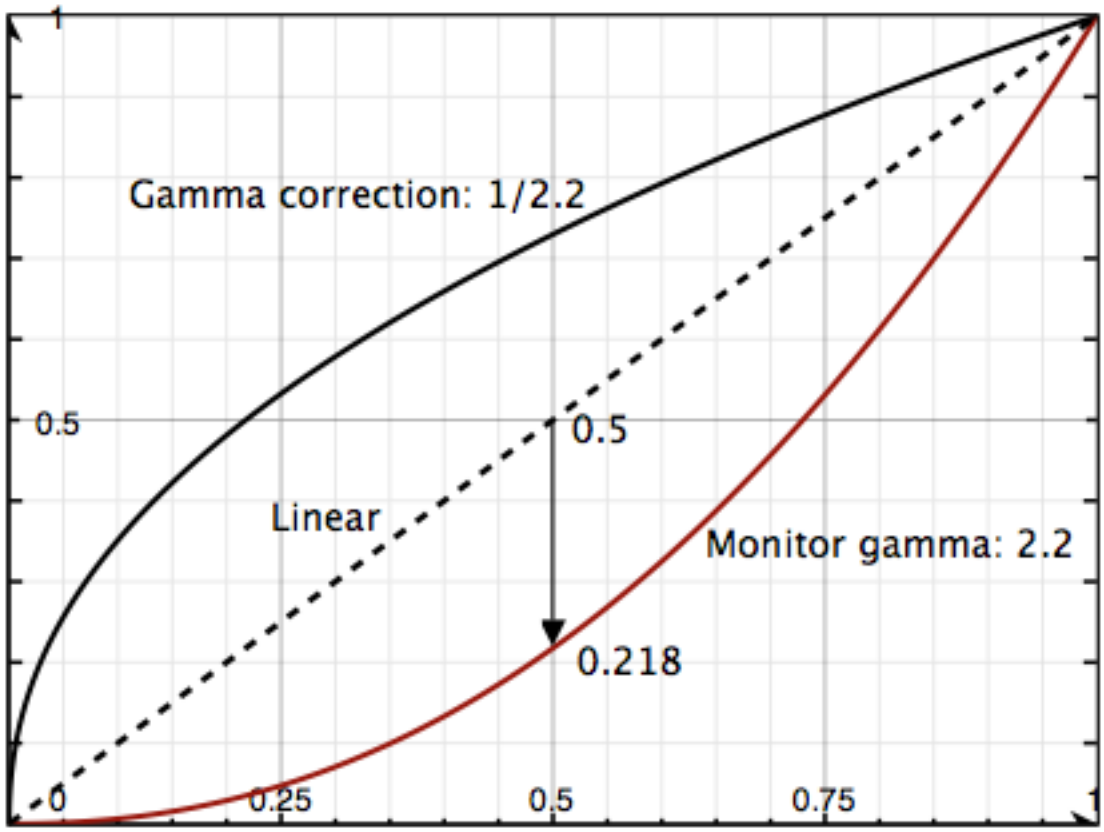
$$x = \frac{X}{X+Y+Z} \quad y = \frac{Y}{X+Y+Z}$$

$$X = \frac{Y}{y}x \quad Z = \frac{Y}{y}(1-x-y)$$



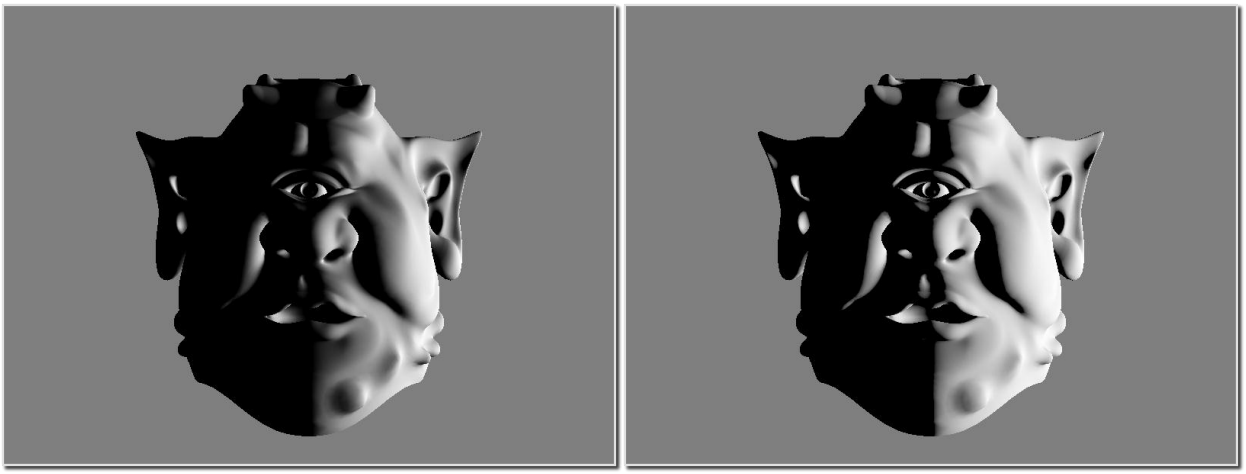


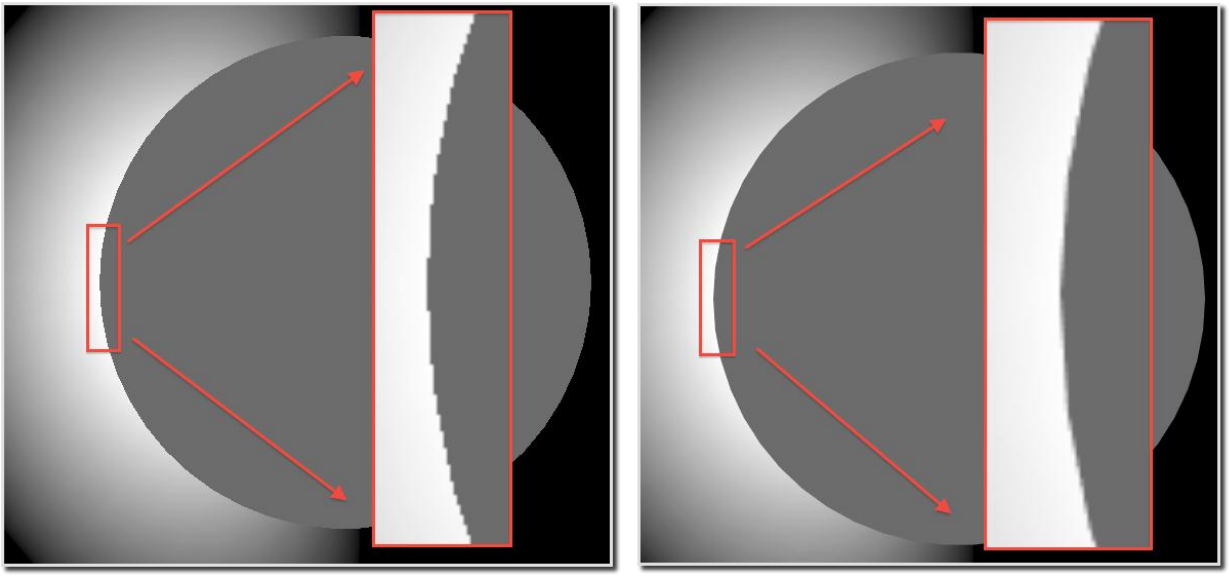

Bright-pass filter
and downsample
Blur
Tone map
Magnify
(linear filter)
+

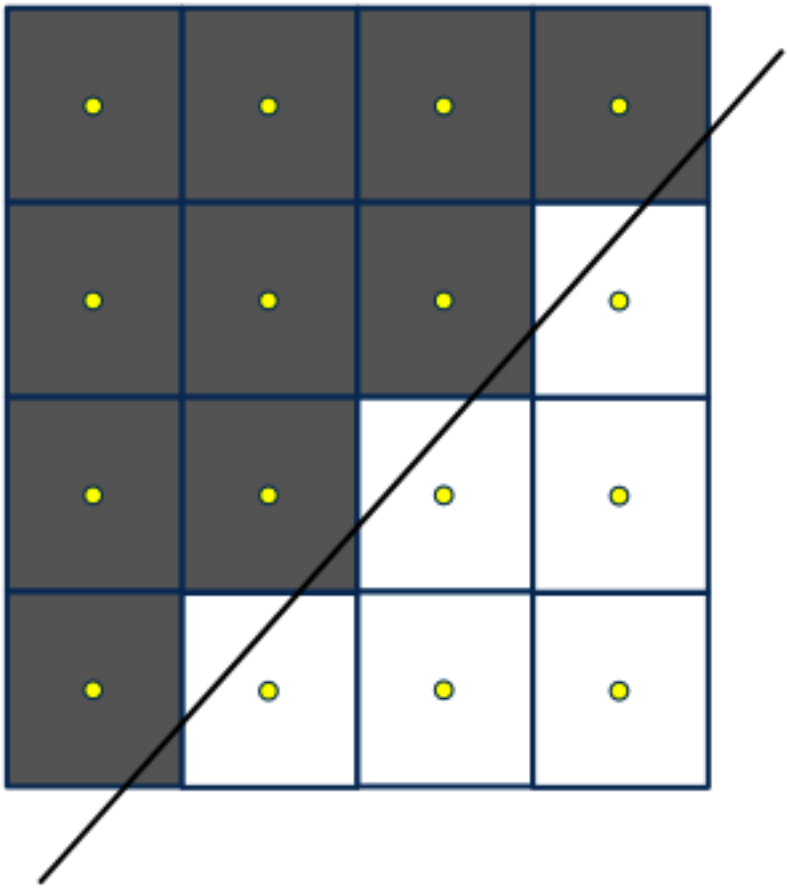

1
Gamma correction: 1/2.2
0.5
0.5
Linear
Monitor gamma: 2.2
0.218
0
0.25
0.5
0.75
1

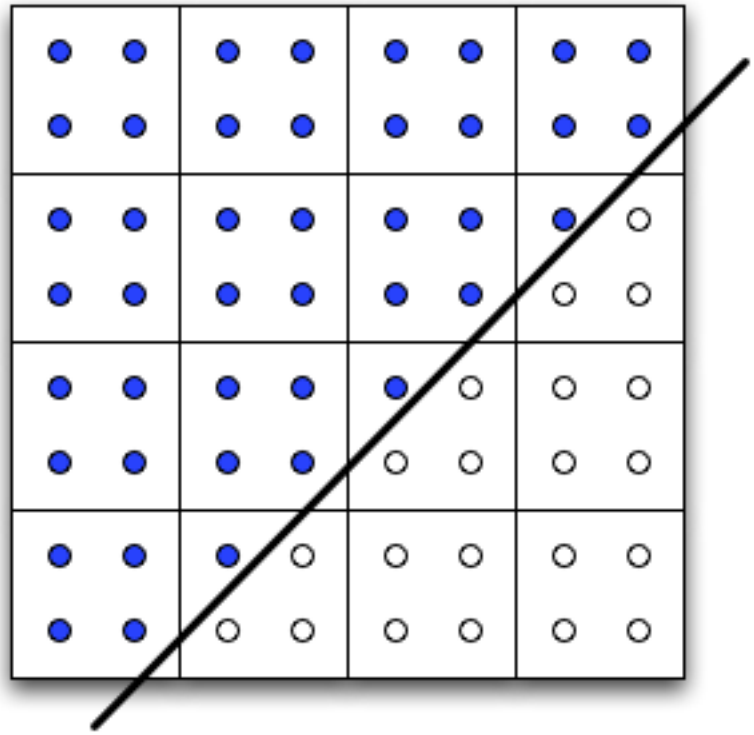
$$P = I^{\gamma}$$

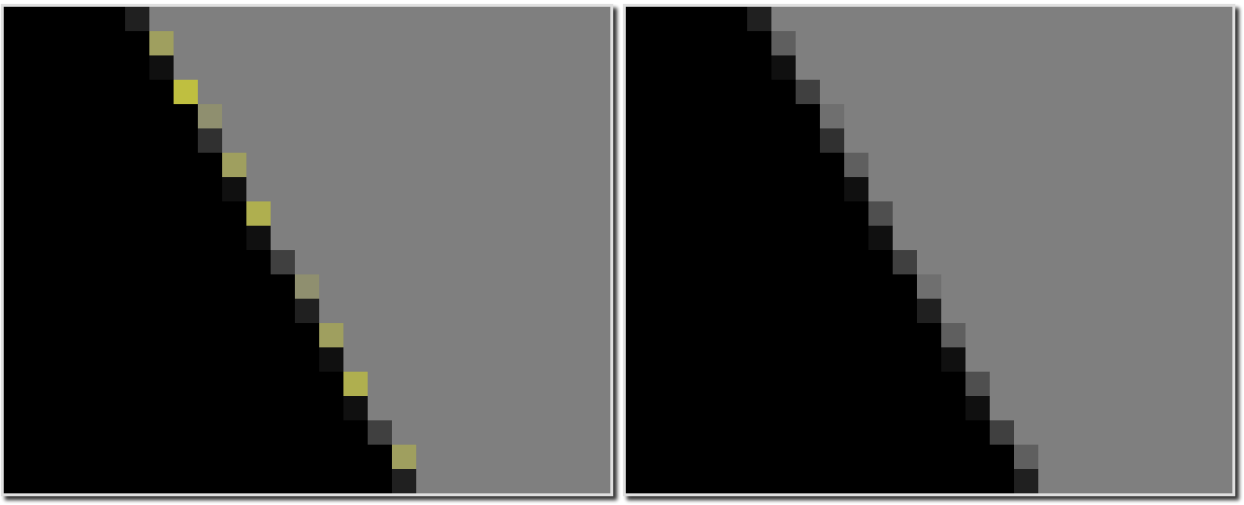
$$I = \left(I^{\frac{1}{\gamma}} \right)^{\gamma}$$

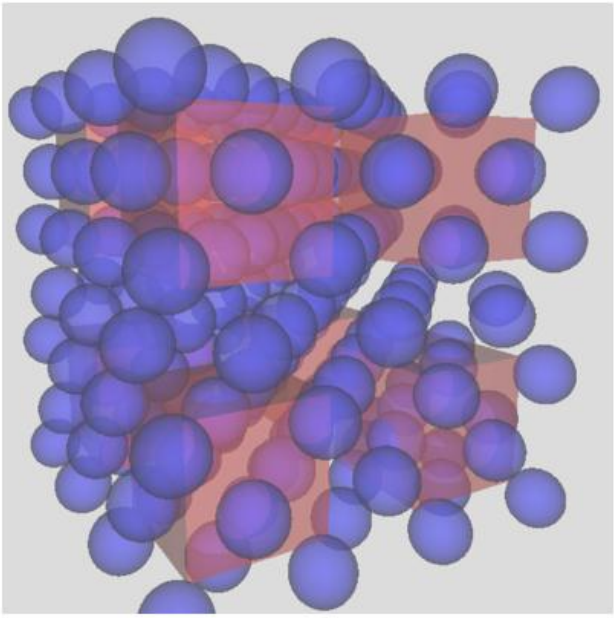


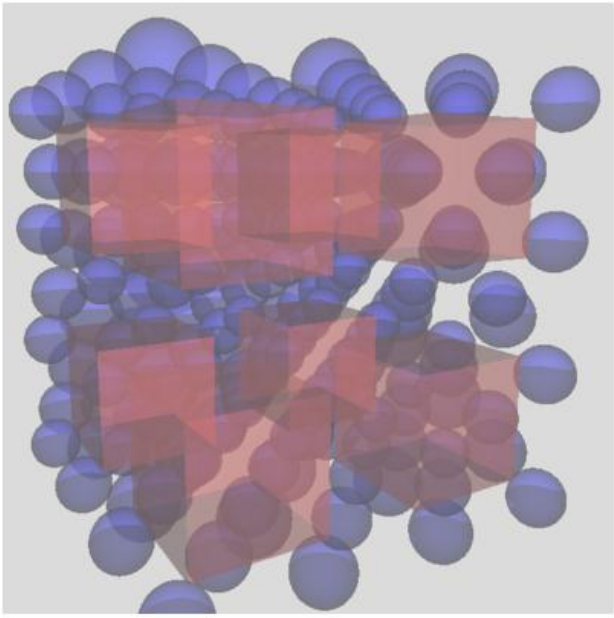












Head Pointer Texture

2	x	x	x
1	x	x	x
0	x	x	x
	0	1	2

Atomic counter

0

Linked-List Buffer

……
……

Head Pointer Texture

	0	1	2
2	x	x	x
1	0	x	x
0	x	x	x

Atomic counter

1

Linked-List Buffer

0	……
Color: ...	……
Depth: 0.75	
Next: x	

Head Pointer Texture

	0	1	2
2	x	x	x
1	1	x	x
0	x	x	x

Atomic counter

2

Linked-List Buffer

0	1	……
Color: ...	Color: ...	……
Depth: 0.75	Depth: 0.5	
Next: x	Next: 0	

Head Pointer Texture

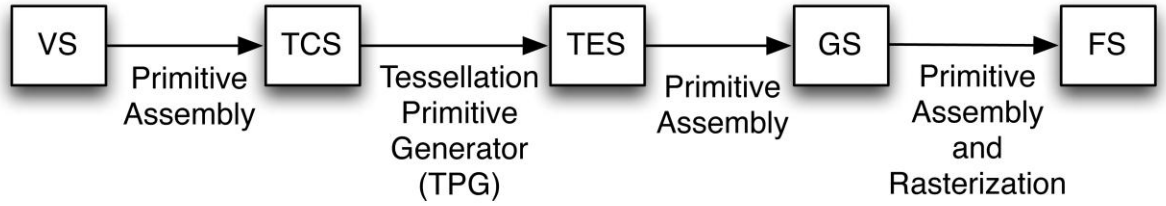
	0	1	2
2	x	x	x
1	3	4	x
0	x	x	x

Atomic counter

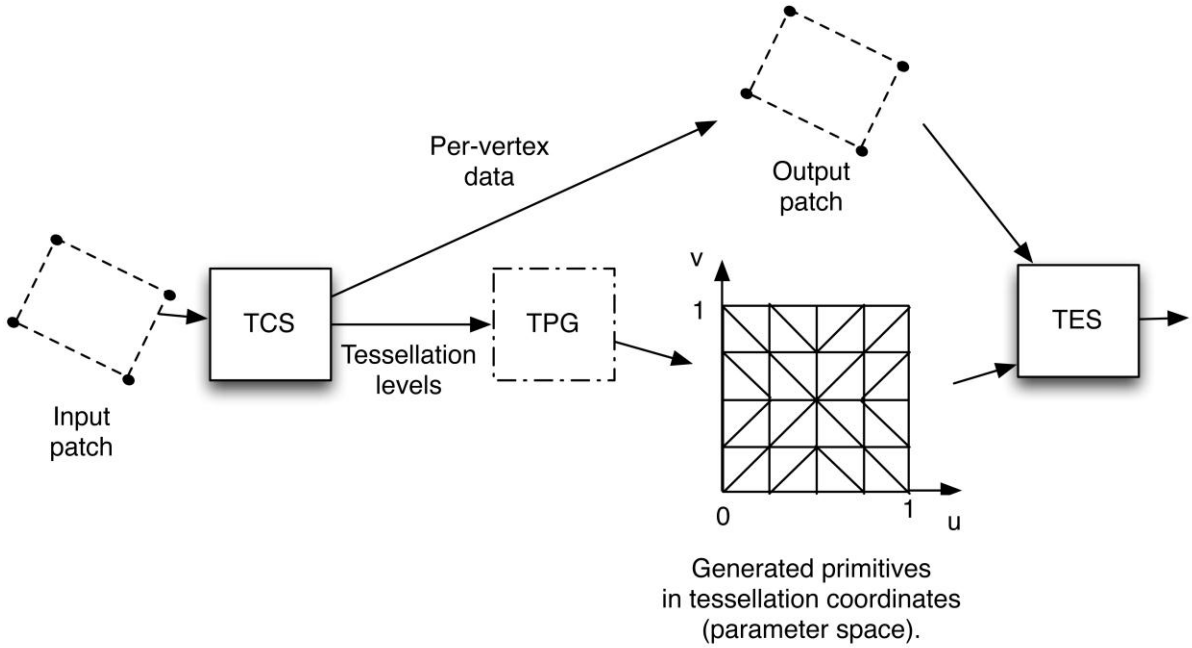
5

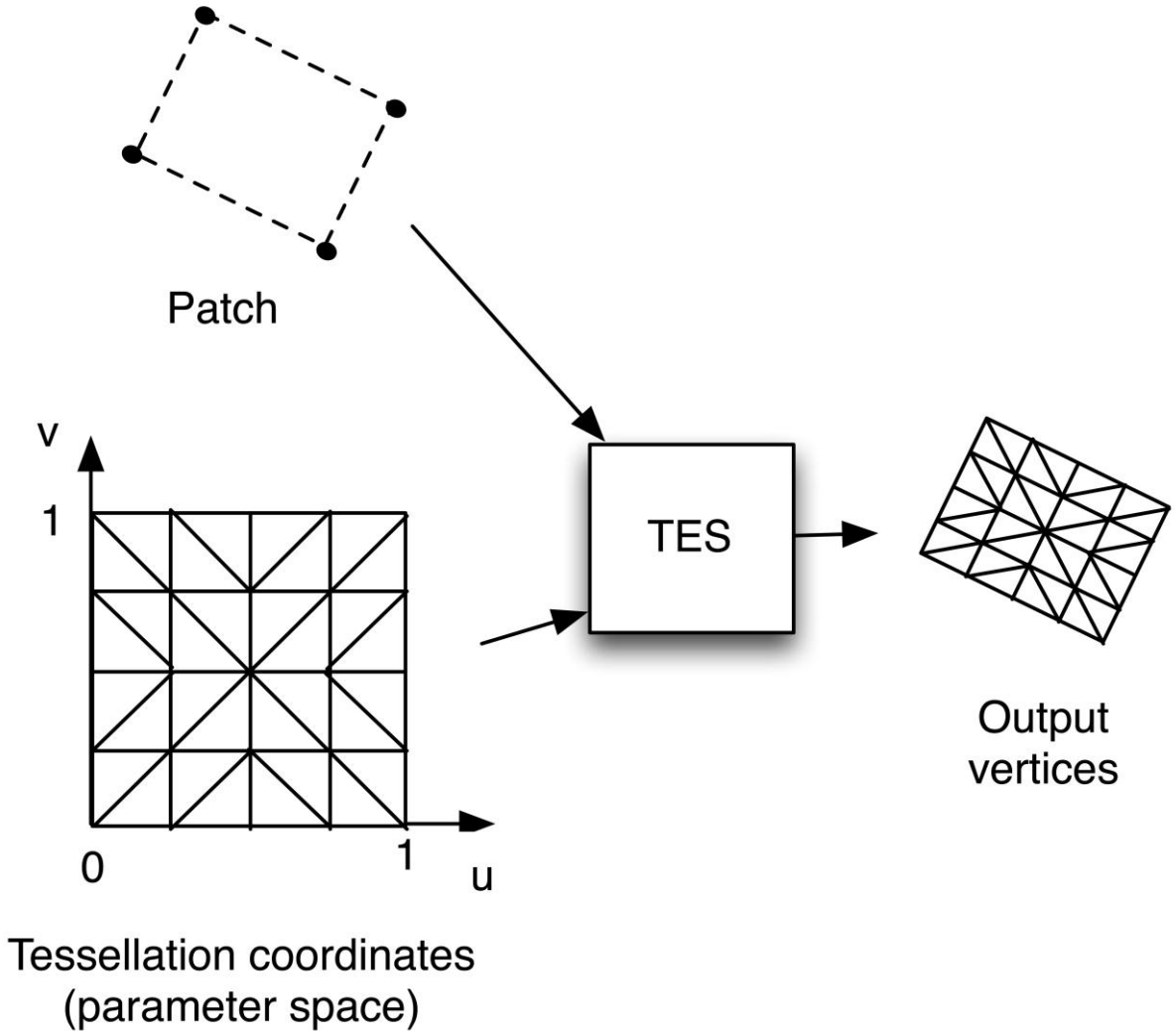
Linked-List Buffer

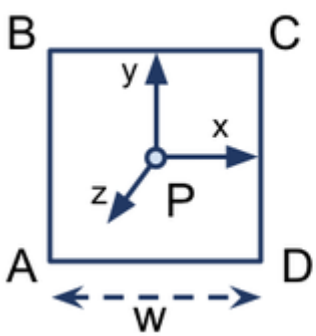
0	1	2	3	4	……
Color: ...	Color: ...	Color: ...	Color: ...	Color: ...	……
Depth: 0.75	Depth: 0.5	Depth: 0.2	Depth: 0.3	Depth: 0.4	
Next: x	Next: 0	Next: x	Next: 1	Next: 2	



VS - Vertex Shader
TCS - Tessellation Control Shader
TES - Tessellation Evaluation Shader
GS - Geometry Shader
FS - Fragment Shader


Per-vertex
data
Output
patch
TCS
TPG
TES
Tessellation
levels
Input
patch
v
1
0
1
u
Generated primitives
in tessellation coordinates
(parameter space).


Patch
v
1
0
1
u
TES
Output
vertices
Tessellation coordinates
(parameter space)

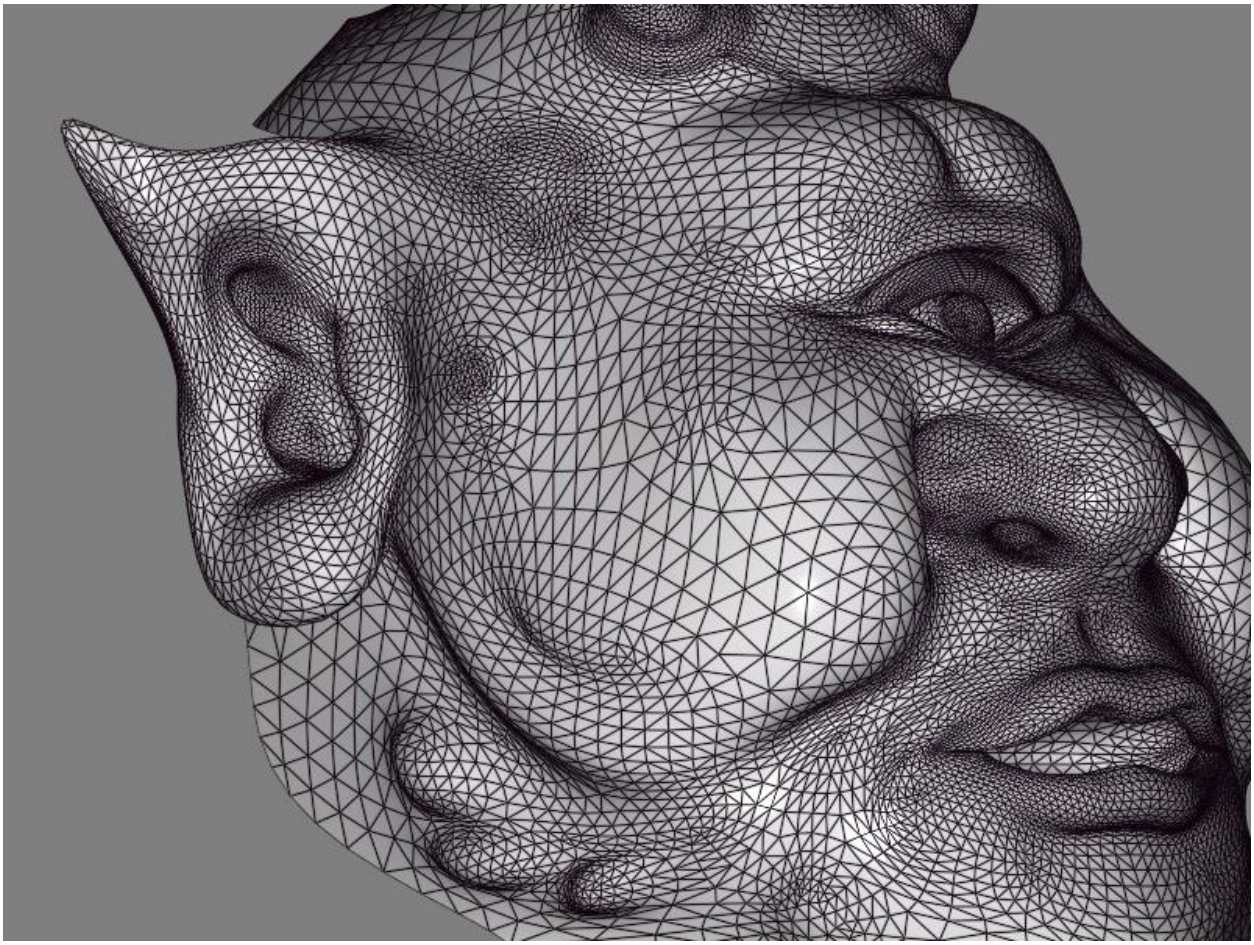


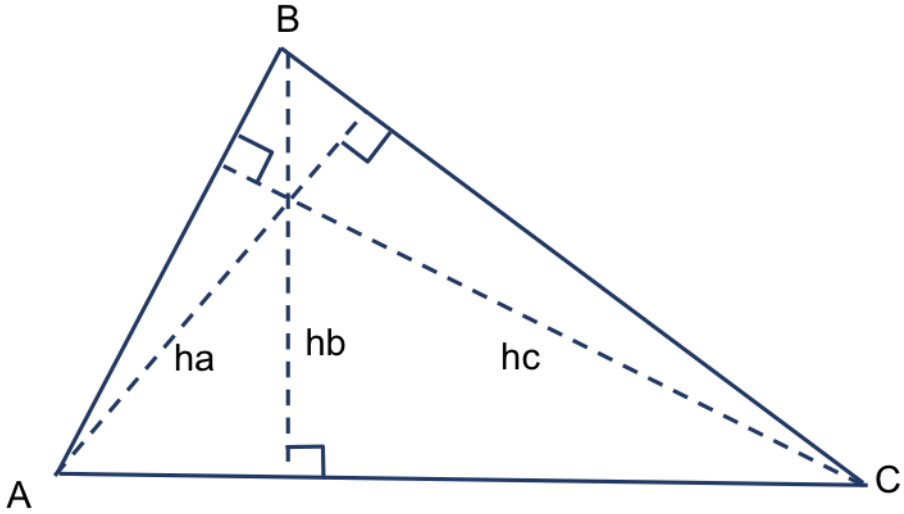
$$A = P + \left(-\frac{w}{2}, -\frac{w}{2}, 0\right)$$

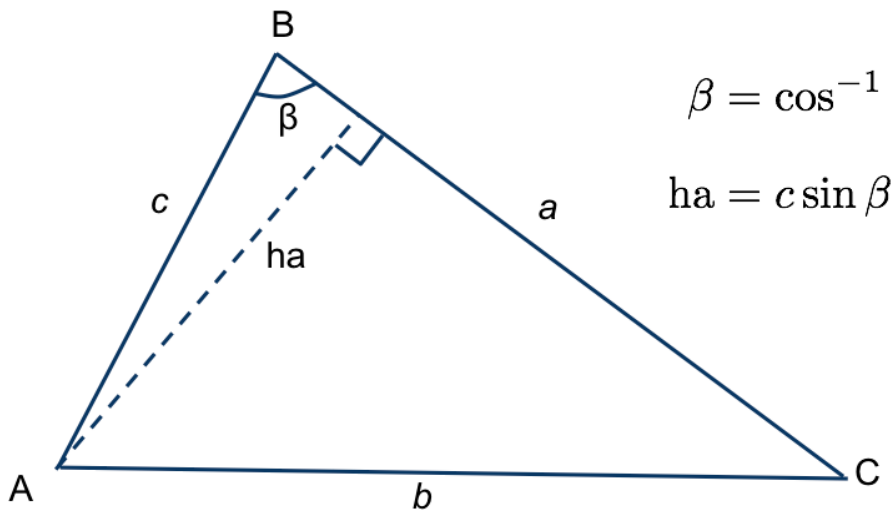
$$B = P + \left(-\frac{w}{2}, \frac{w}{2}, 0\right)$$

$$C = P + \left(\frac{w}{2}, \frac{w}{2}, 0\right)$$

$$D = P + \left(\frac{w}{2}, -\frac{w}{2}, 0\right)$$

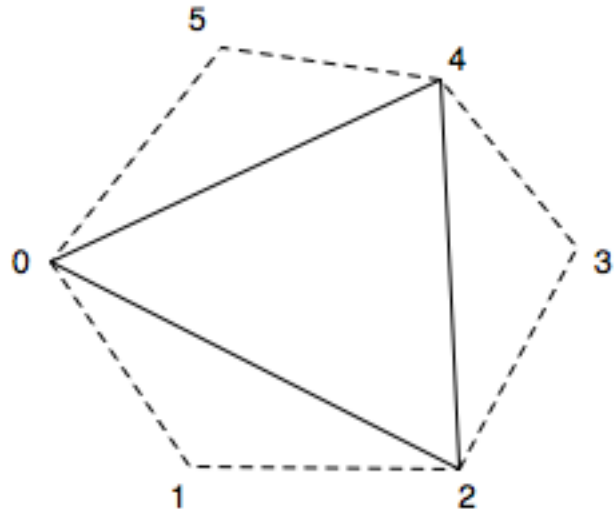



B
ha
hb
hc
A
C

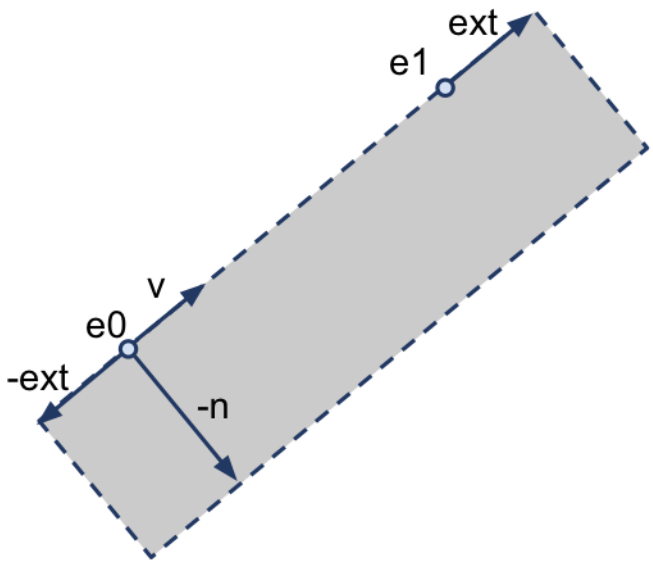


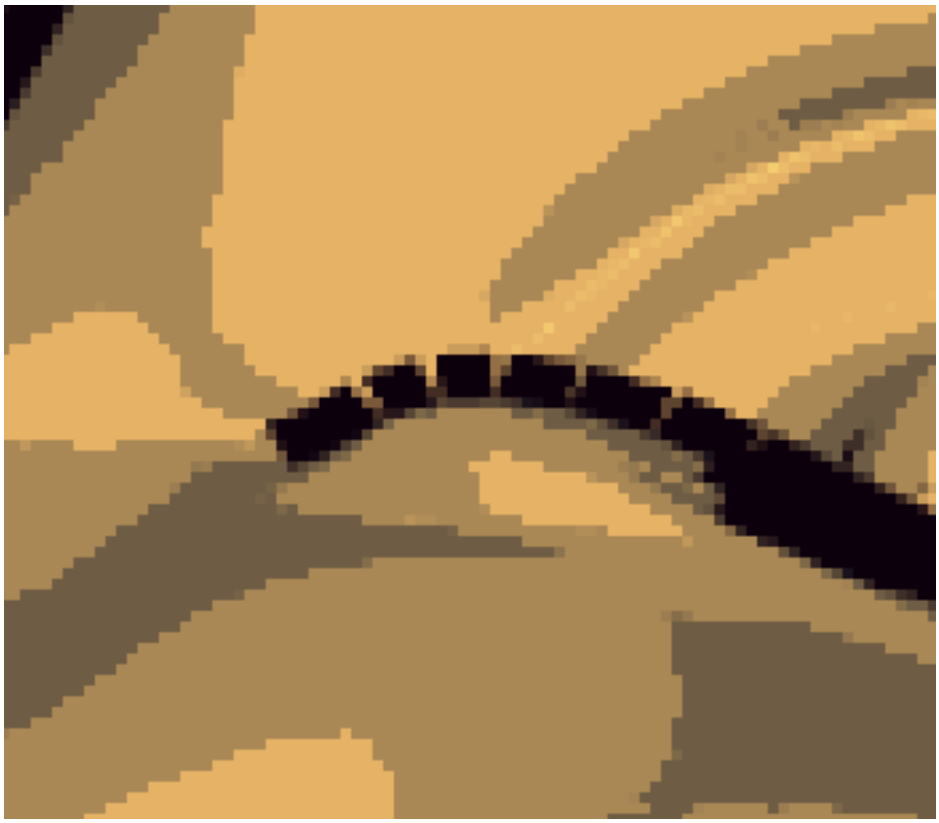
$$\beta = \cos^{-1}\left(\frac{a^2 + c^2 - b^2}{2ac}\right)$$

$$\mathrm{ha} = c \sin \beta$$



$$n_z = (A_x B_y - B_x A_y) + (B_x C_y - C_x B_y) + (C_x A_y - A_x C_y)$$





$$B_i^n(t) = \binom{n}{i} (1-t)^{n-i} t^i$$

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}$$

$$P(t) = \sum_{i=0}^{n} B_i^n(t) P_i$$

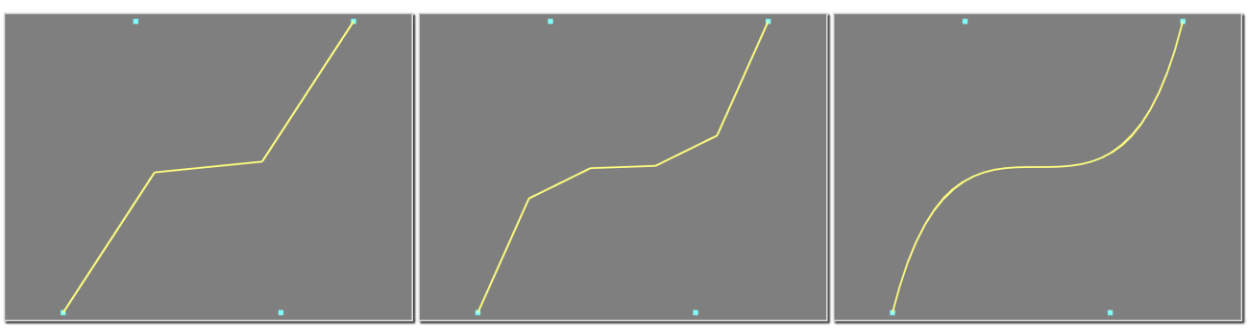
$$P(t) = B_0^3(t)P_0 + B_1^3(t)P_1 + B_2^3(t)P_2 + B_3^3(t)P_3$$

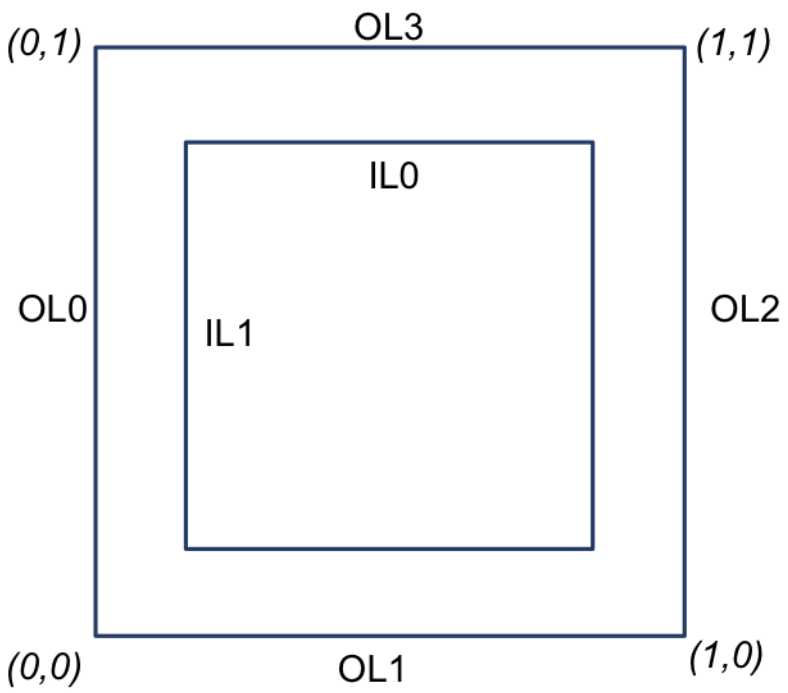
$$B_0^3(t) = (1-t)^3$$

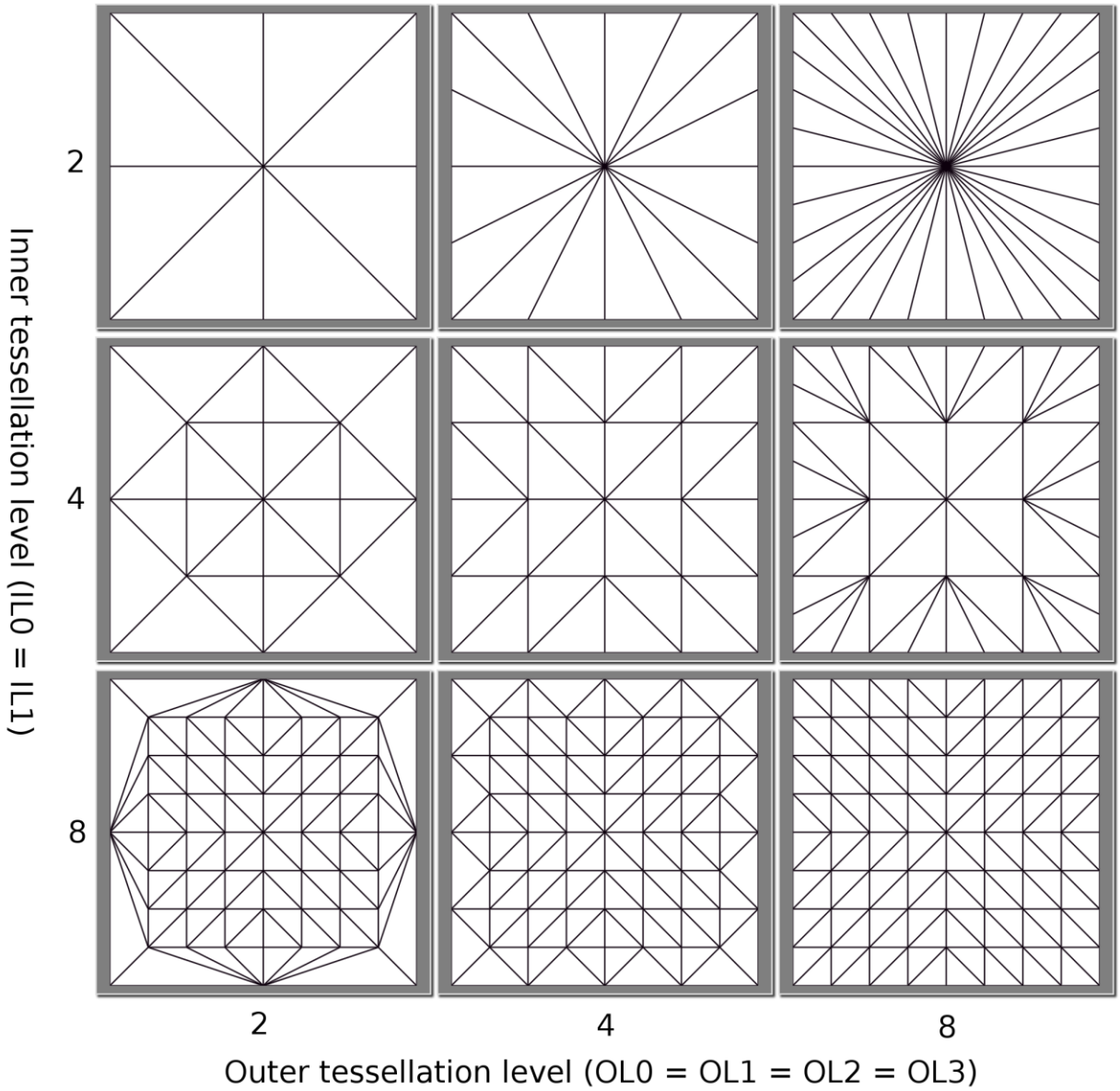
$$B_1^3(t) = 3(1-t)^2 t$$

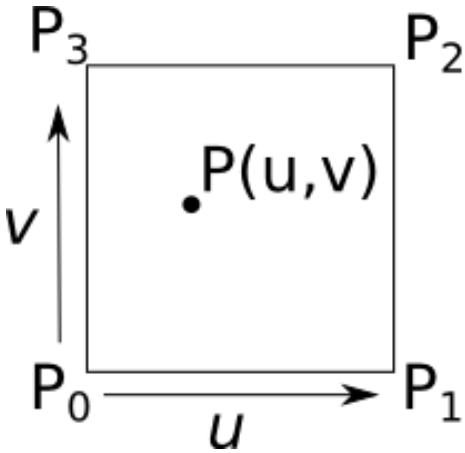
$$B_2^3(t) = 3(1-t)t^2$$

$$B_3^3(t) = t^3$$

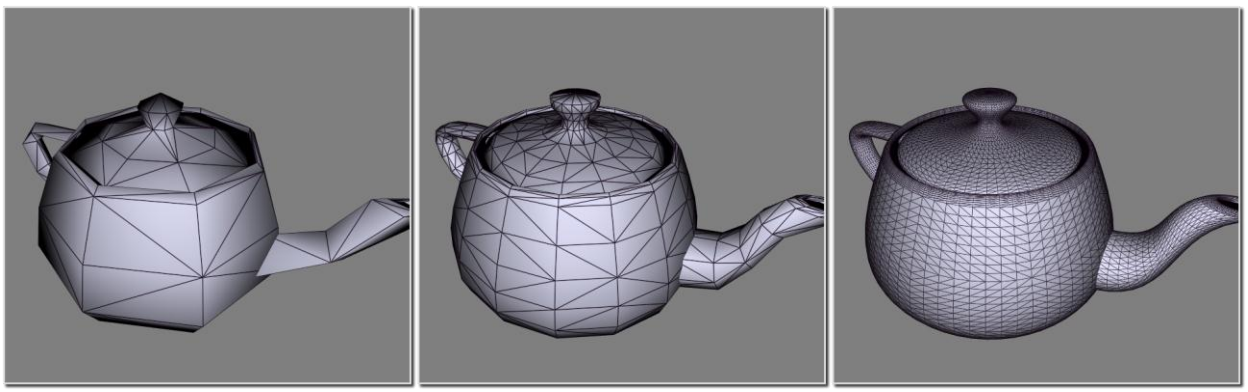



(0,1)
OL3
(1,1)
IL0
OL0
IL1
OL2
(0,0)
OL1
(1,0)


Inner tessellation level (IL0 = IL1)
2
4
8
2
4
8
Outer tessellation level (OL0 = OL1 = OL2 = OL3)



$$
\begin{aligned}
P(u,v) = & P_0(1-u)(1-v)+ \\
& P_1 u(1-v)+ \\
& P_2 uv+ \\
& P_3 v(1-u)
\end{aligned}
$$

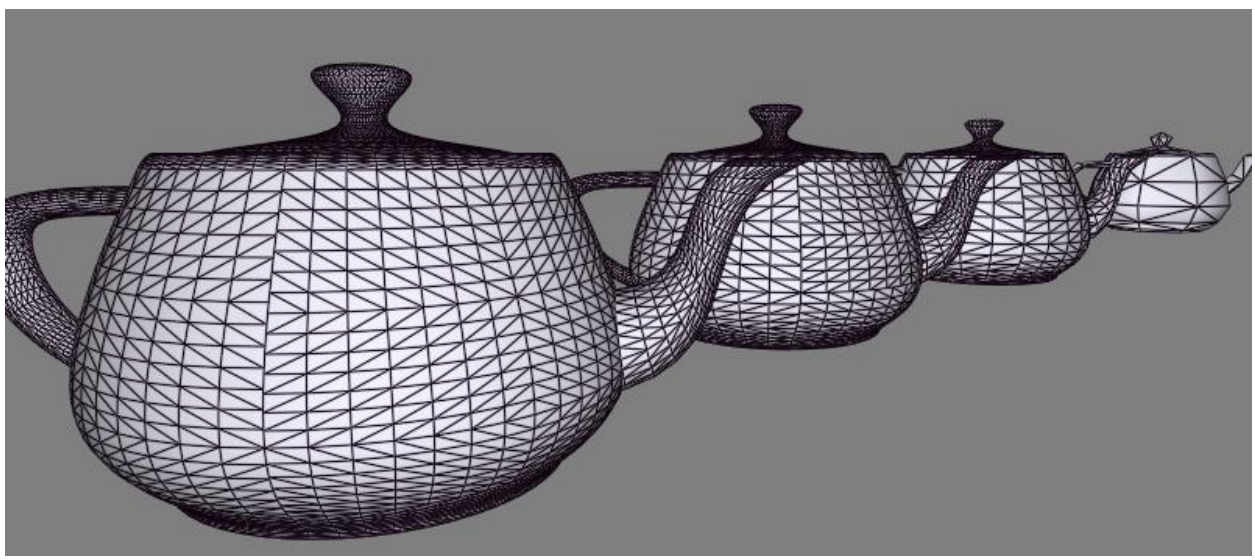


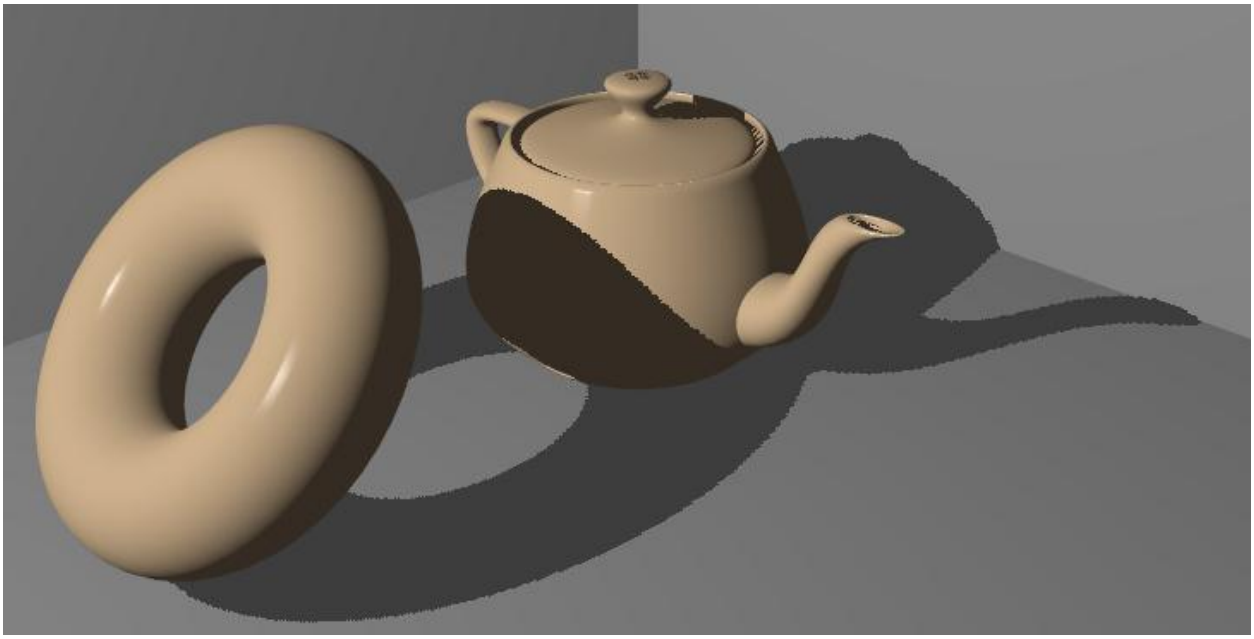
$$
P(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} B_i^3(u) B_j^3(v) P_{ij}
$$

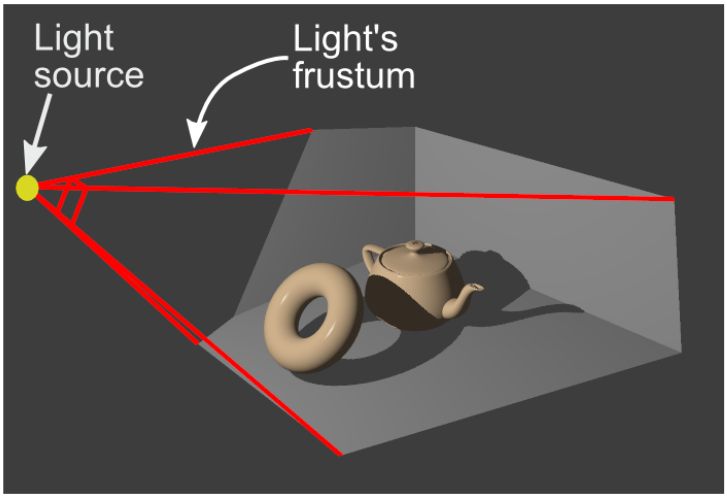
$$
\mathbf{n}(u,v) = \frac{\partial P}{\partial u} \times \frac{\partial P}{\partial v}
$$

$$\frac{\partial P}{\partial u} = \sum_{i=0}^{3} \sum_{j=0}^{3} \frac{\partial B_i^3(u)}{\partial u} B_j^3(v) P_{ij}$$

$$\frac{\partial P}{\partial v} = \sum_{i=0}^{3} \sum_{j=0}^{3} B_i^3(u) \frac{\partial B_j^3(v)}{\partial v} P_{ij}$$





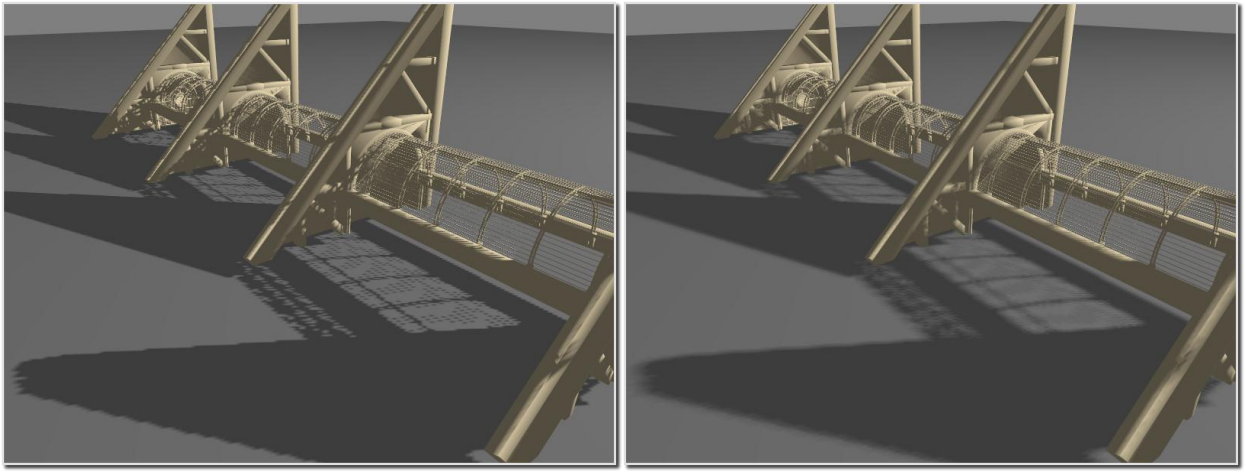

Light
source
Light's
frustum

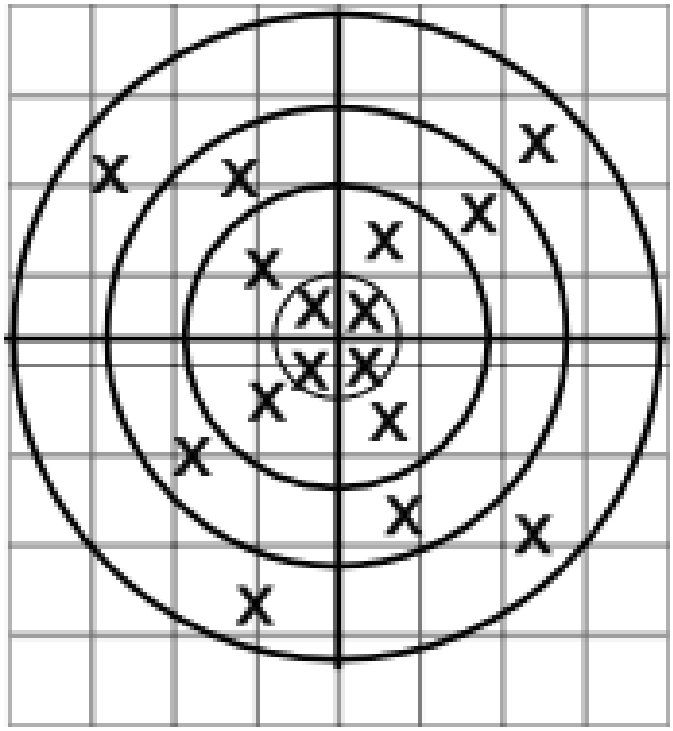
$$\mathbf{B} = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

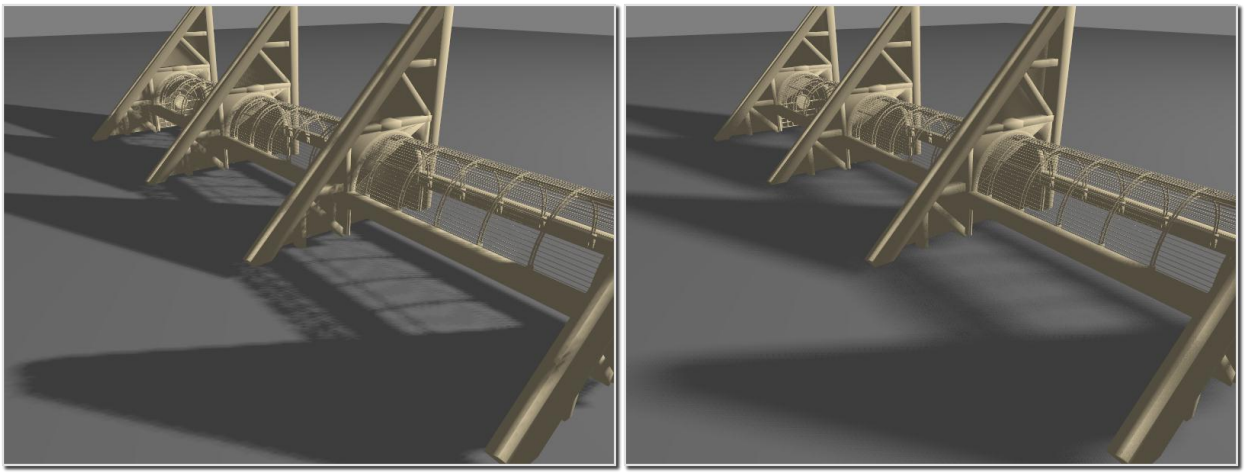
$$\mathbf{Q} = \mathbf{B}\mathbf{P}_l\mathbf{V}_l\mathbf{W}$$

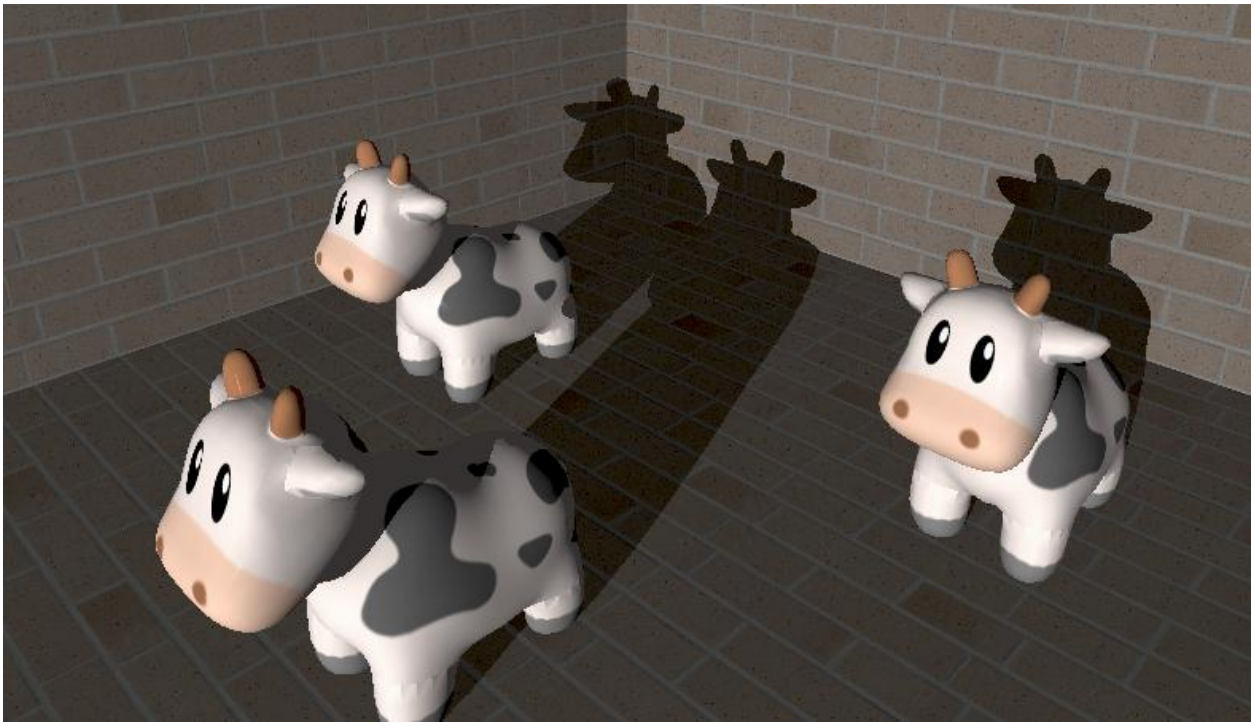
$$\mathbf{Q} = \mathbf{S}\mathbf{C}$$

$$\mathbf{S} = \mathbf{B}\mathbf{P}_l\mathbf{V}_l\mathbf{M}$$



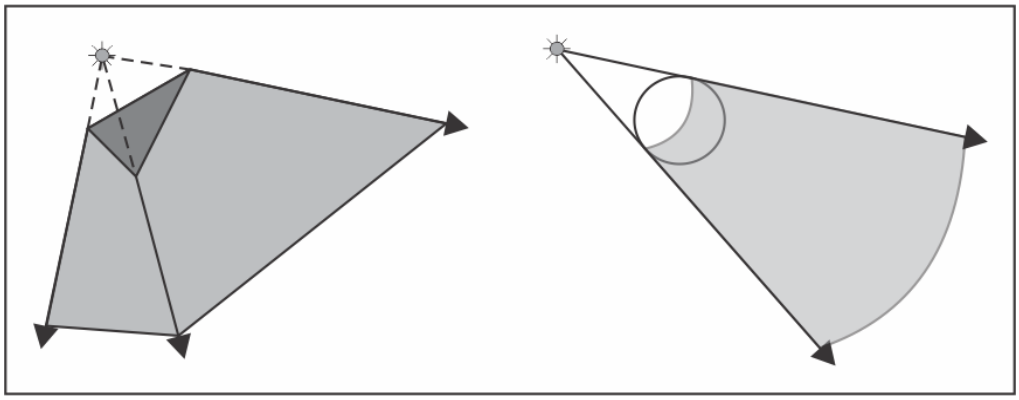


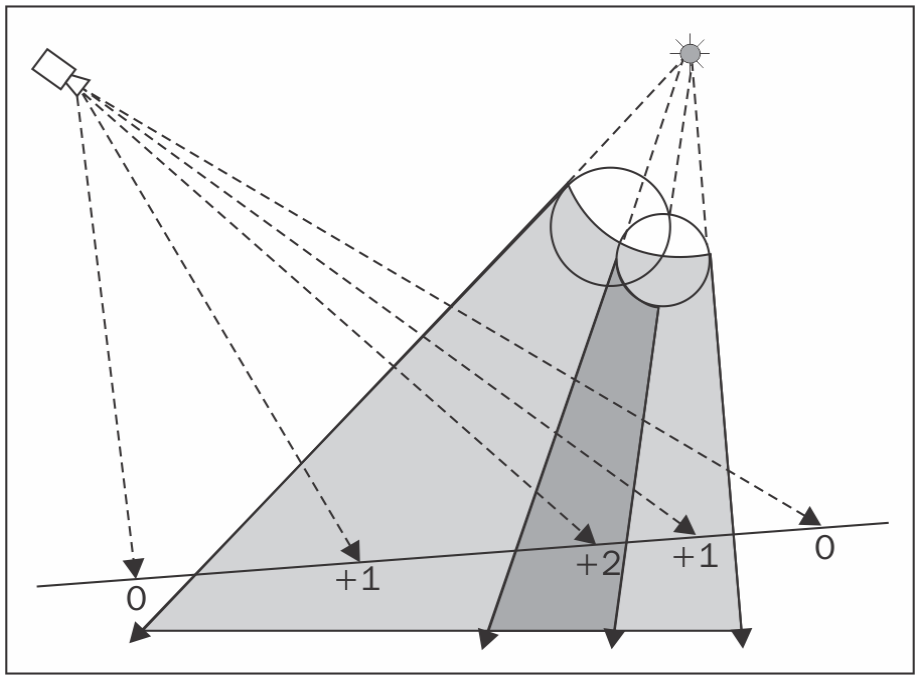


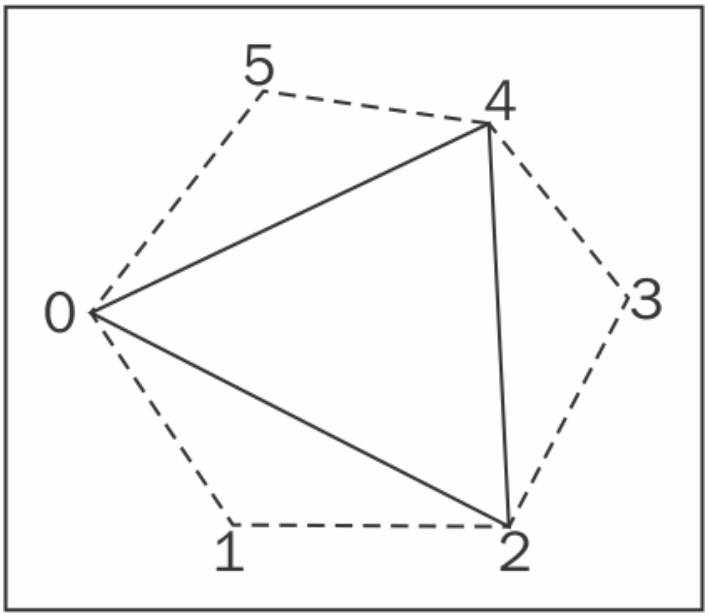


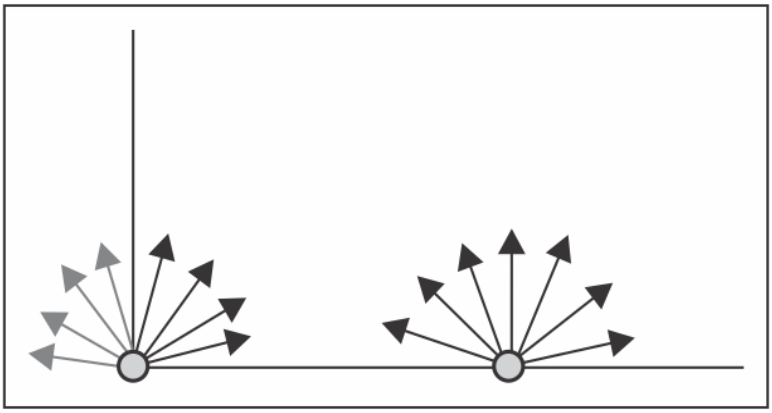
$$w_x = \sqrt{v}\cos(2\pi u)$$

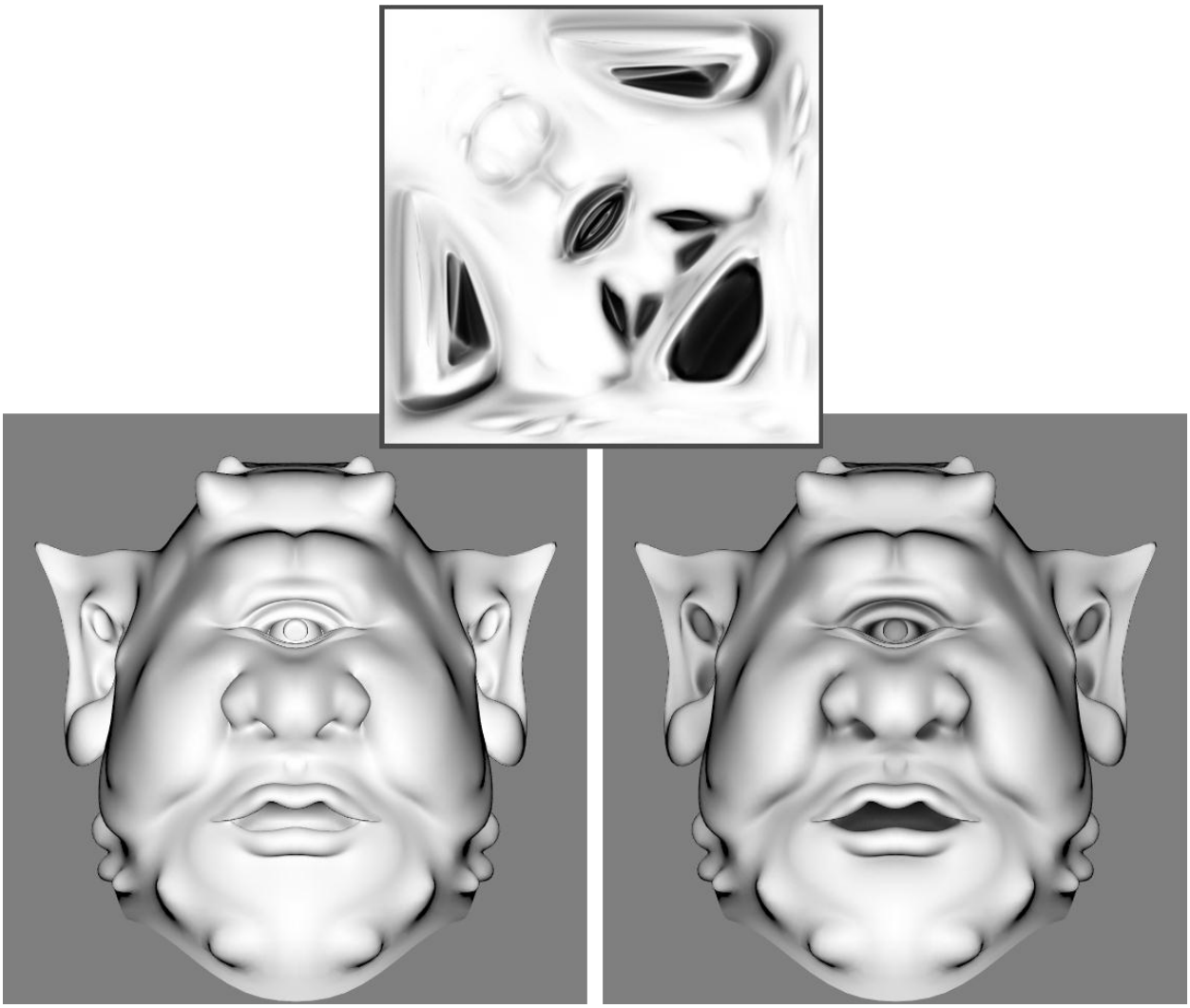
$$w_y = \sqrt{v}\sin(2\pi u)$$

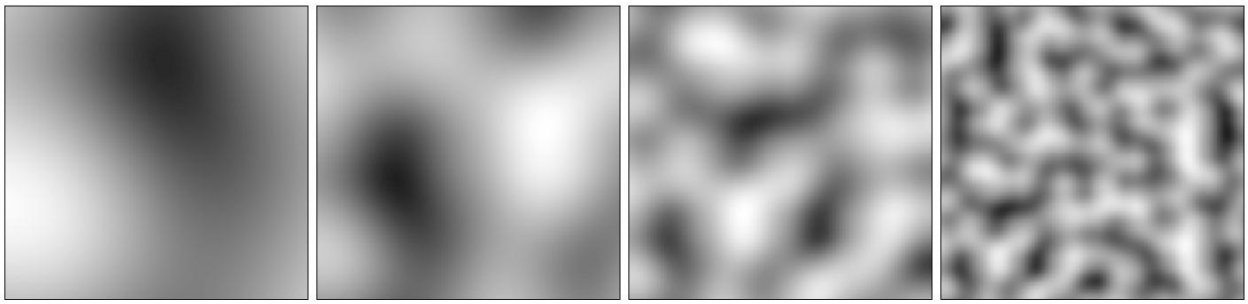



0
+1
+2
+1
0


5
4
0
3
1
2

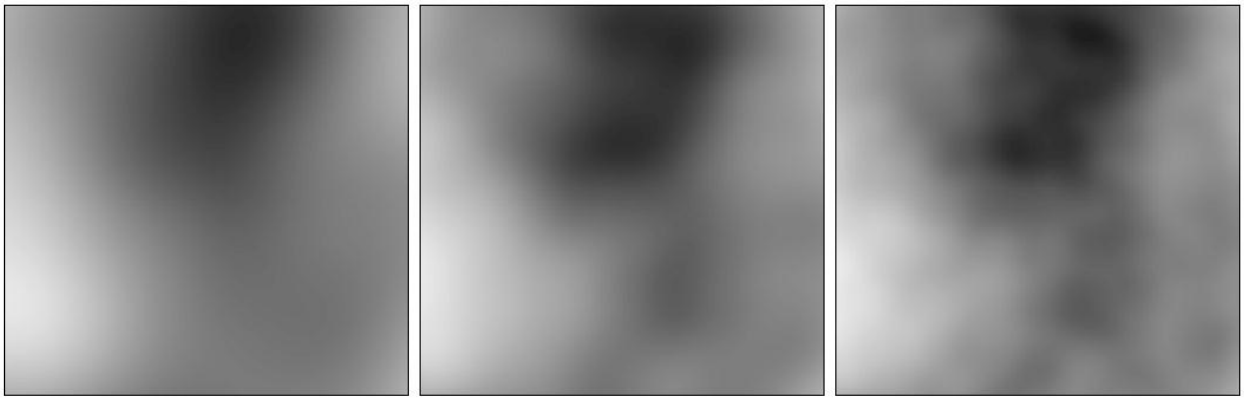


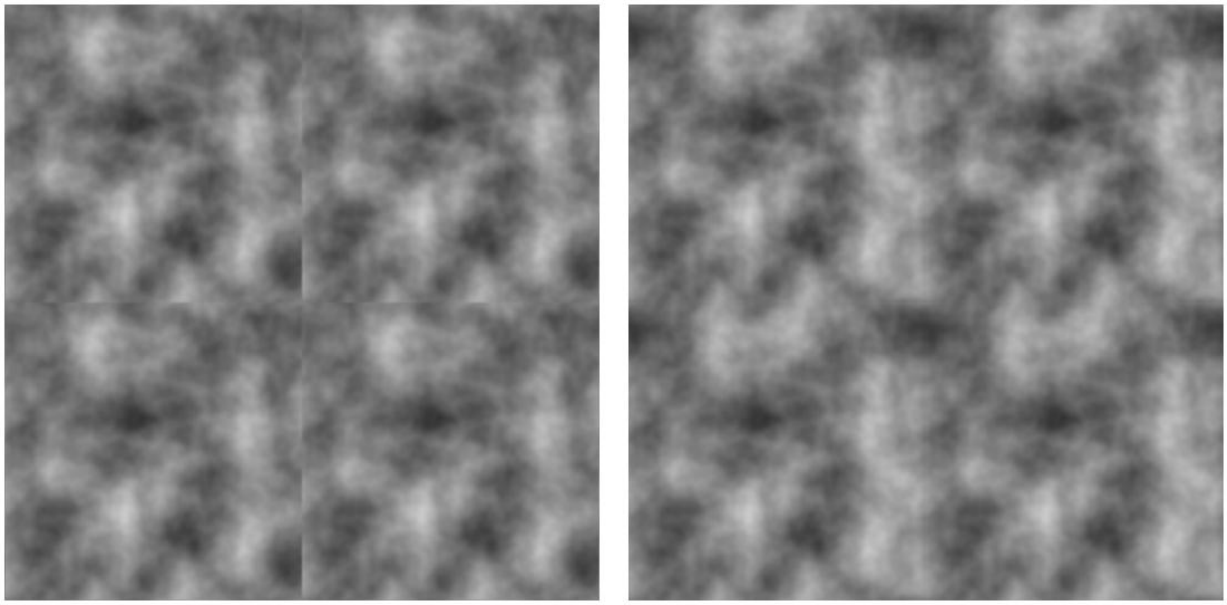


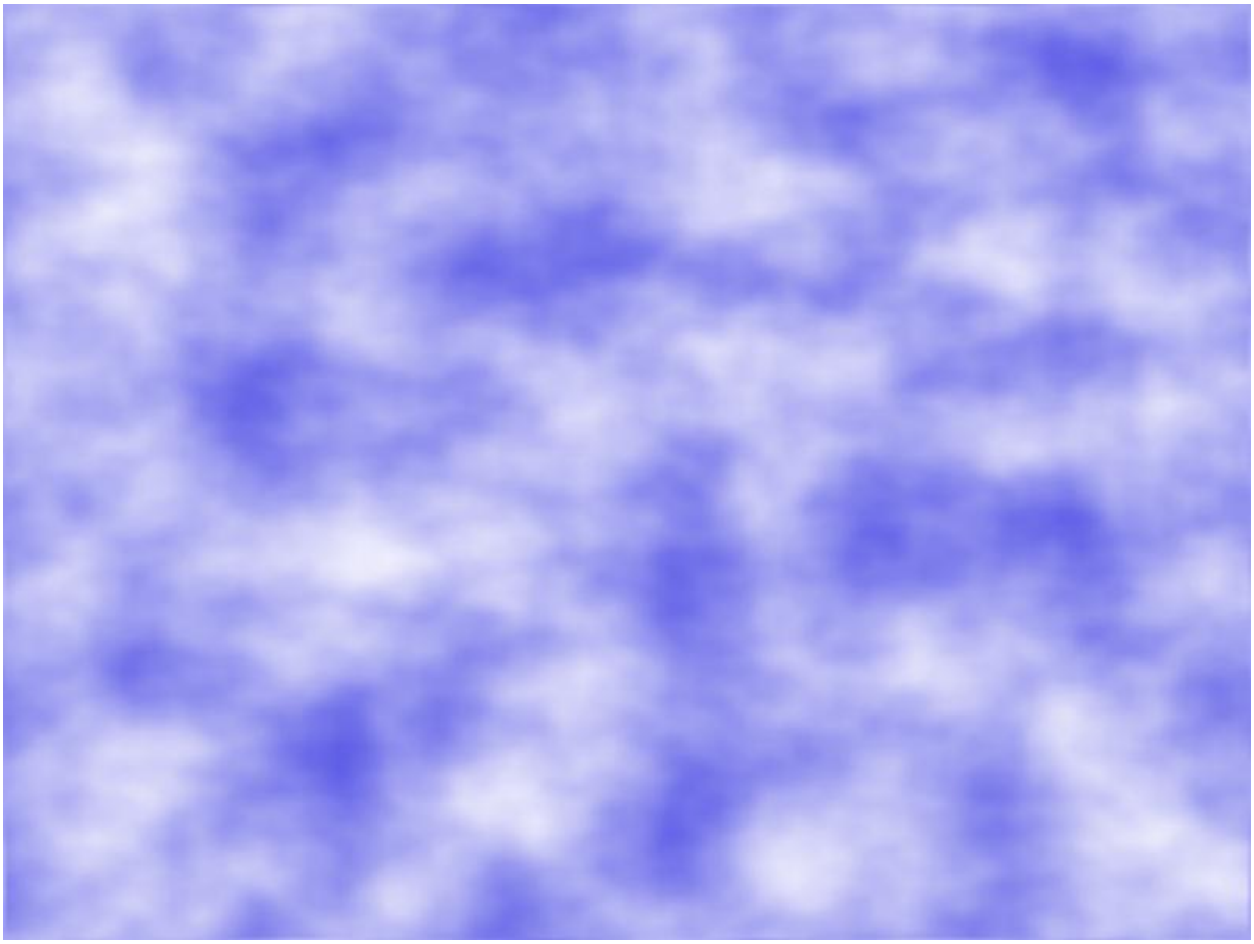


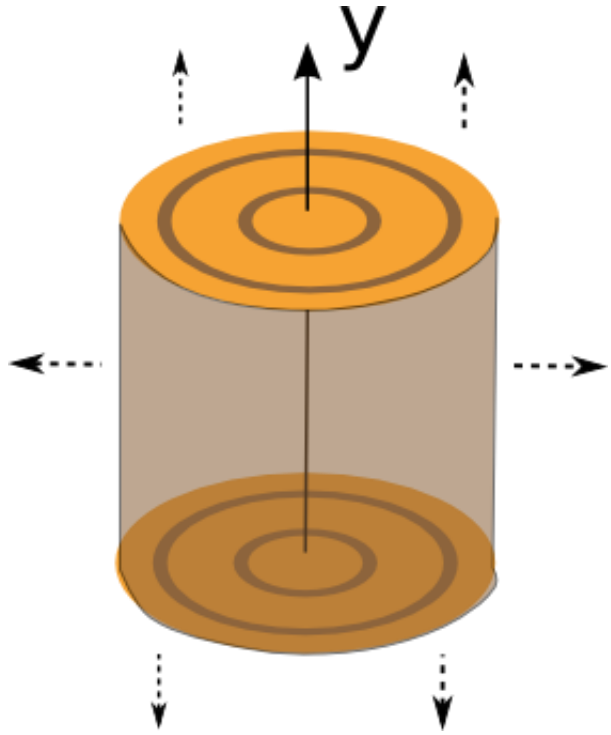
$$n_i(x, y) = P(2^i x, 2^i y)$$

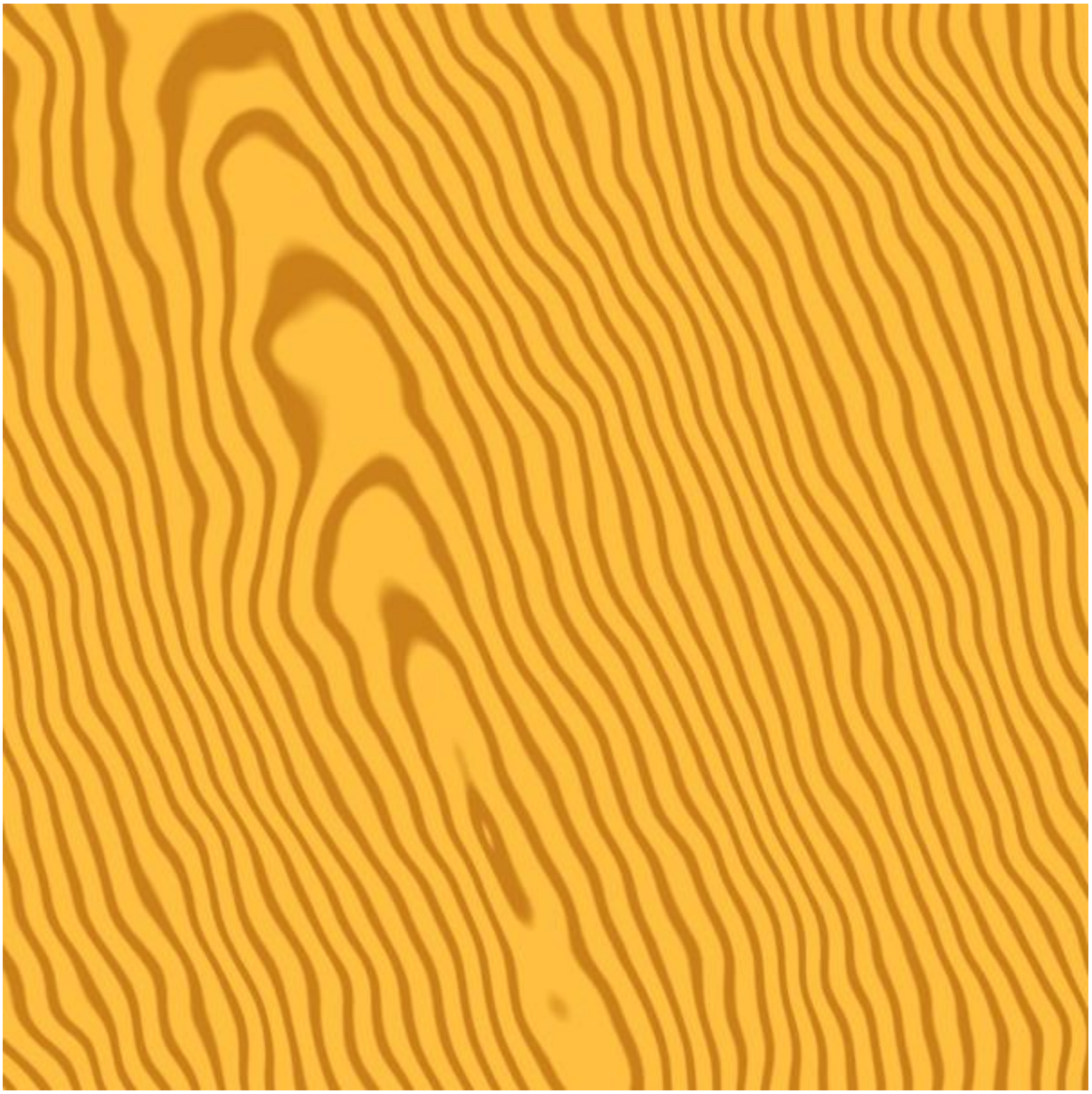
$$n(x,y) = \sum_{i=0}^{N-1} \frac{P(2^i ax, 2^i ay)}{b^i}$$

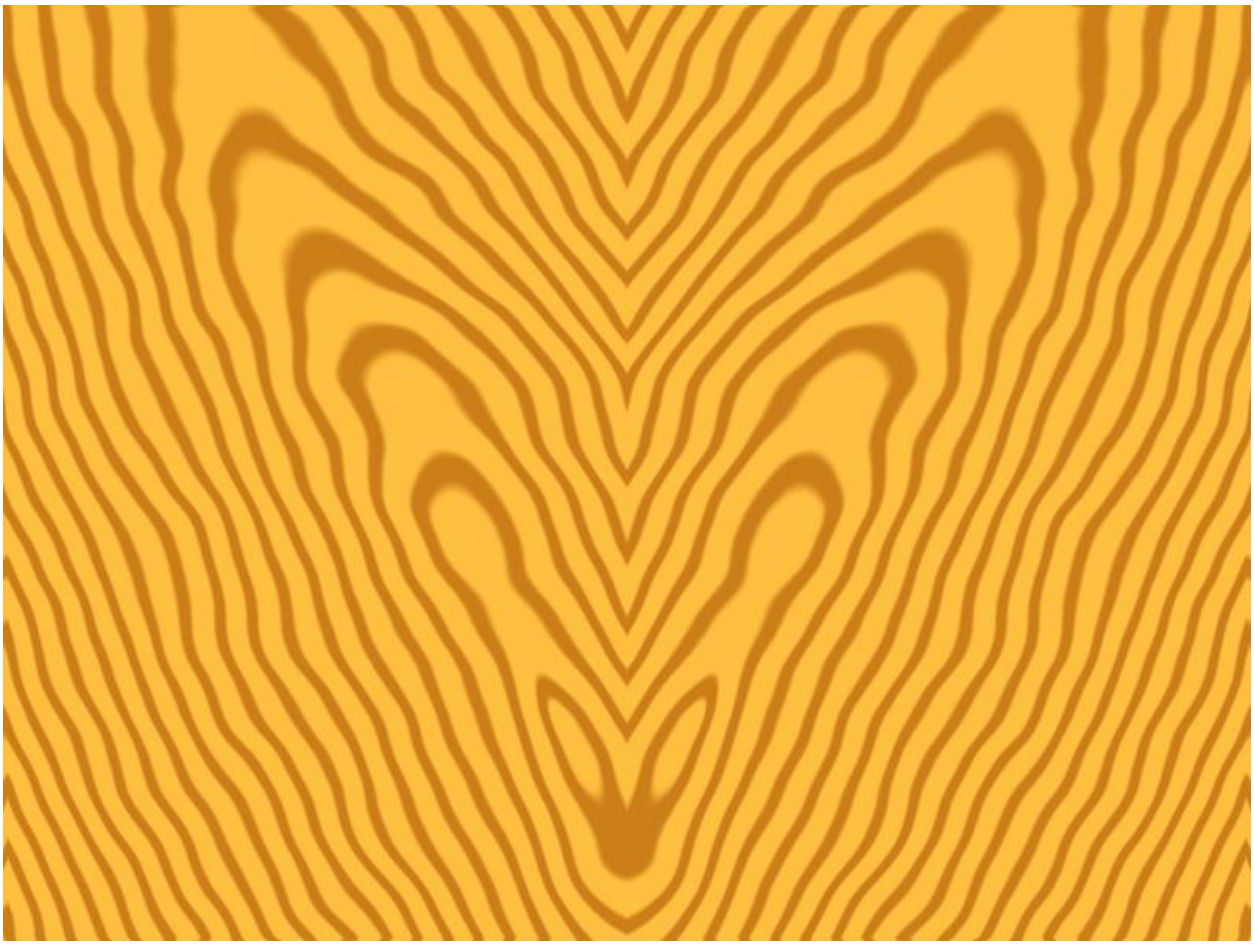


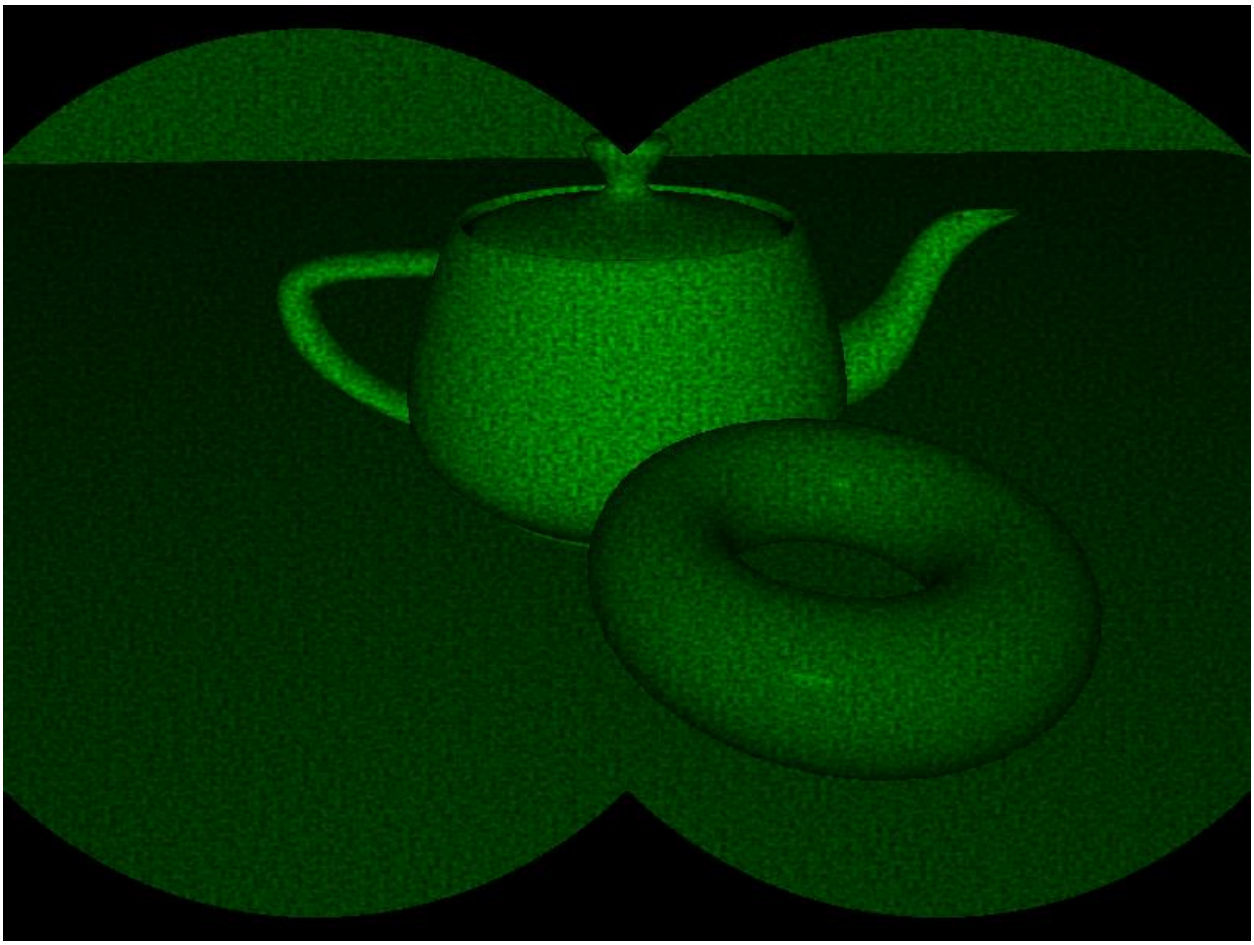


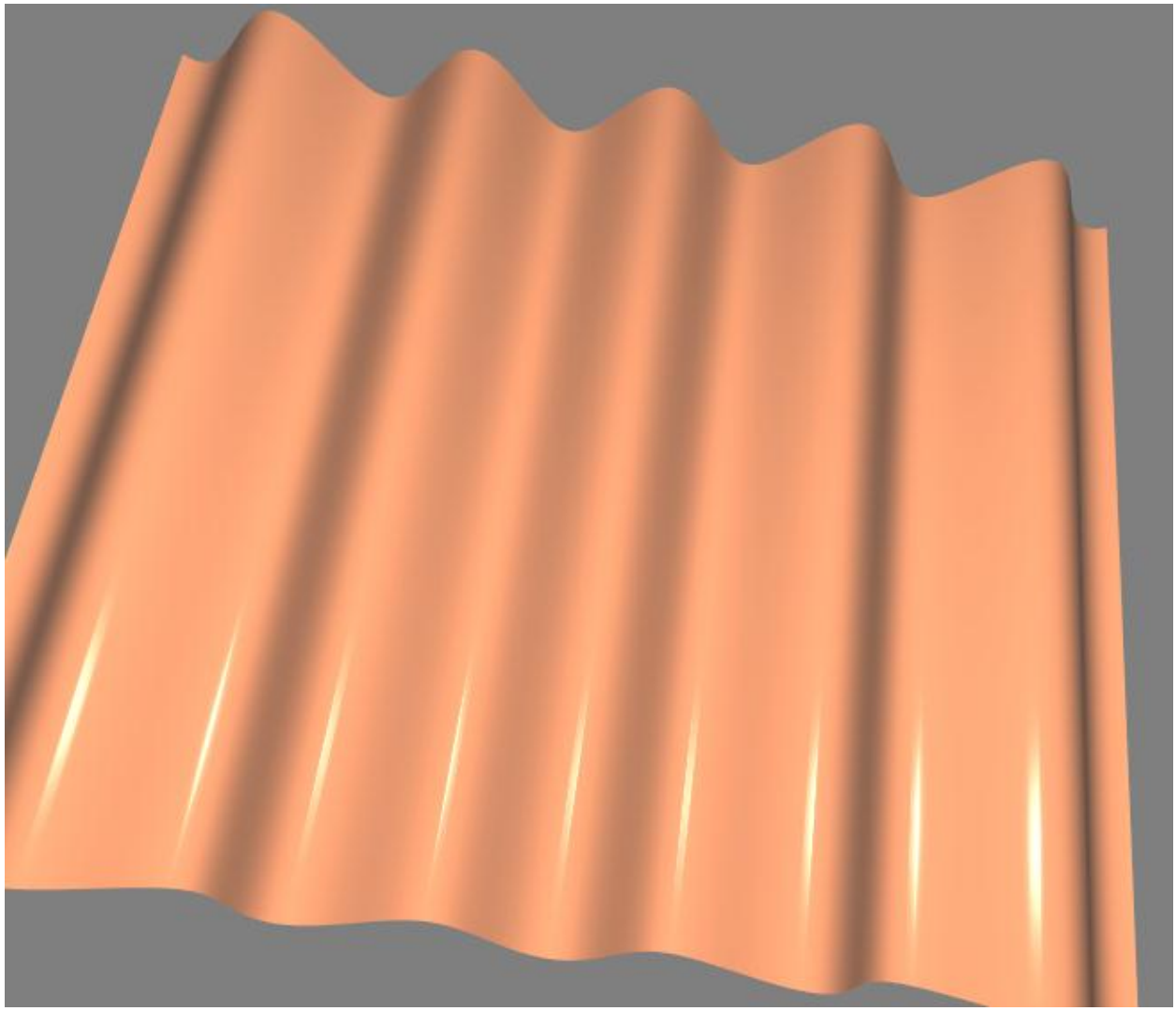


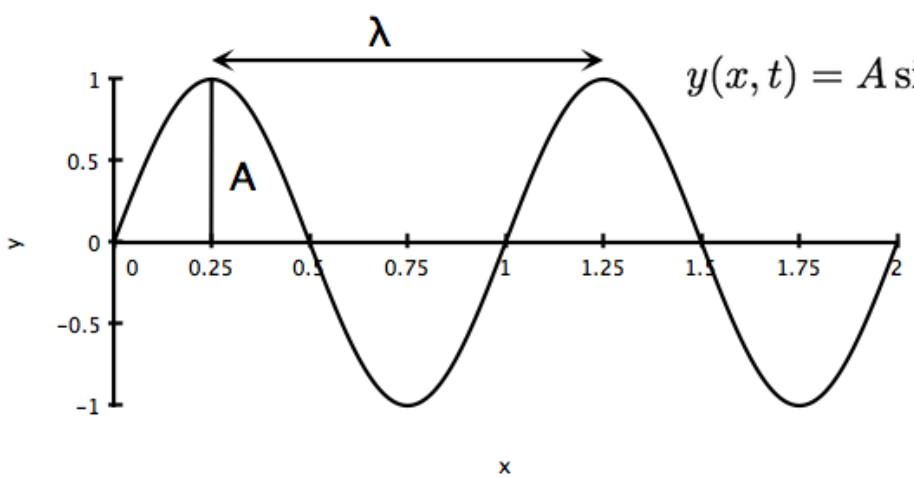

y











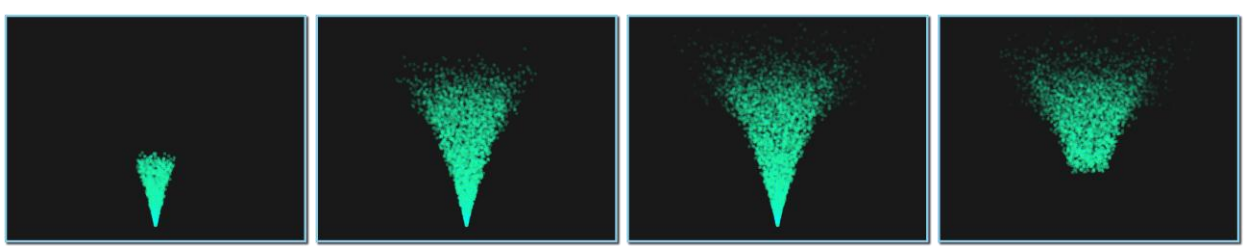
$$y(x,t) = A\sin\left(\frac{2\pi}{\lambda}(x - vt)\right)$$

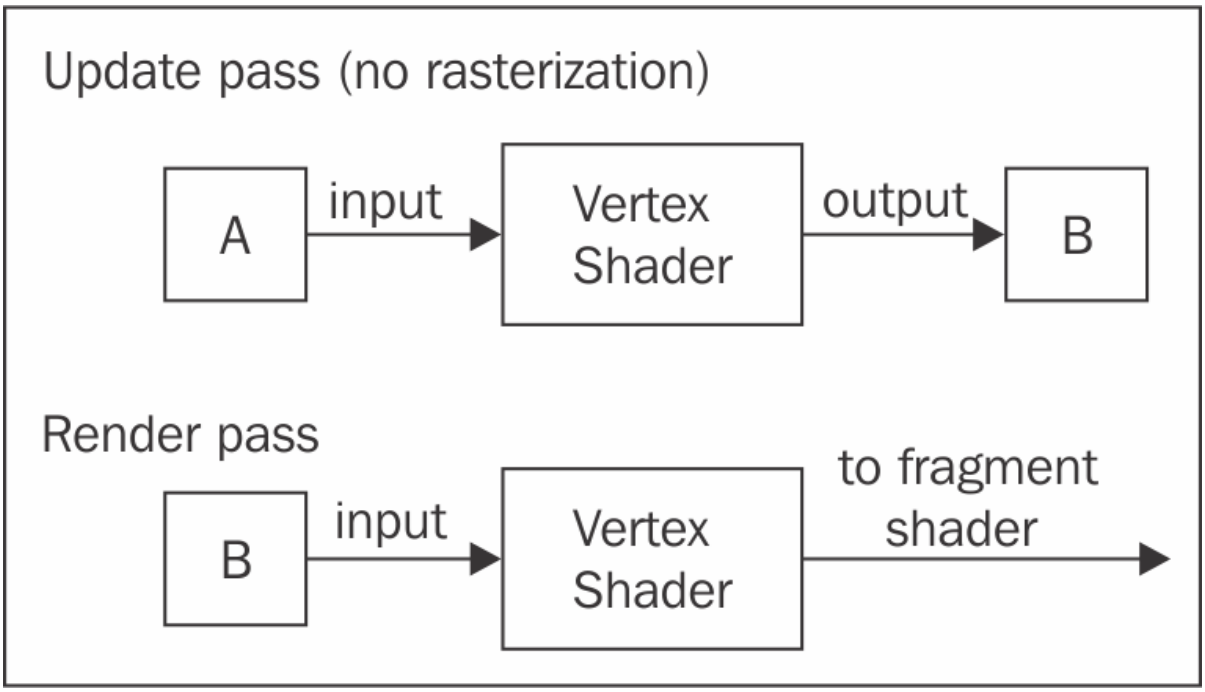
$$\mathbf{n}(x,t) = \left(-A\frac{2\pi}{\lambda}\cos\left(\frac{2\pi}{\lambda}(x-vt)\right), 1\right)$$

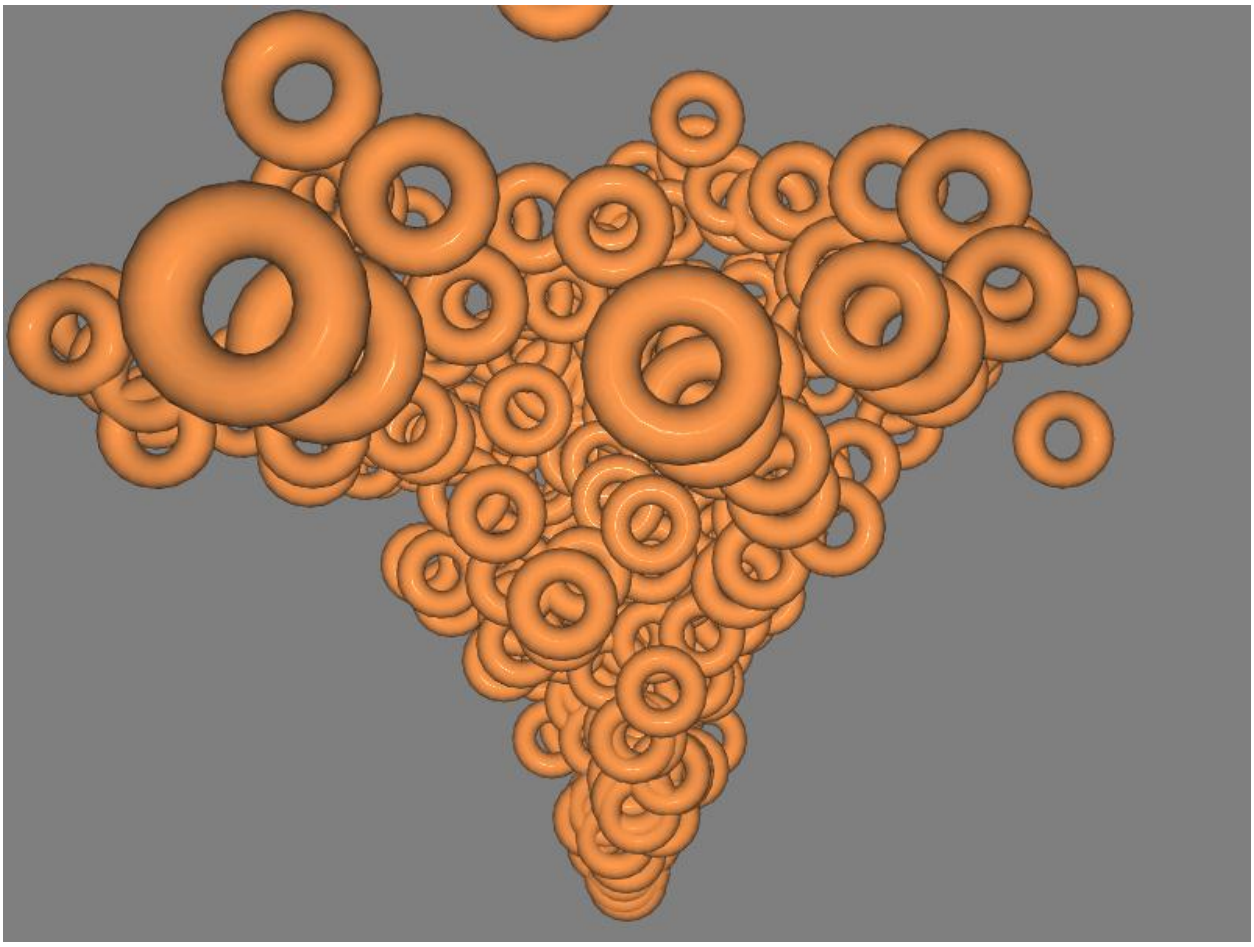
$$P(t) = P_0 + \mathbf{v}_0 t + \frac{1}{2}\mathbf{a}t^2$$

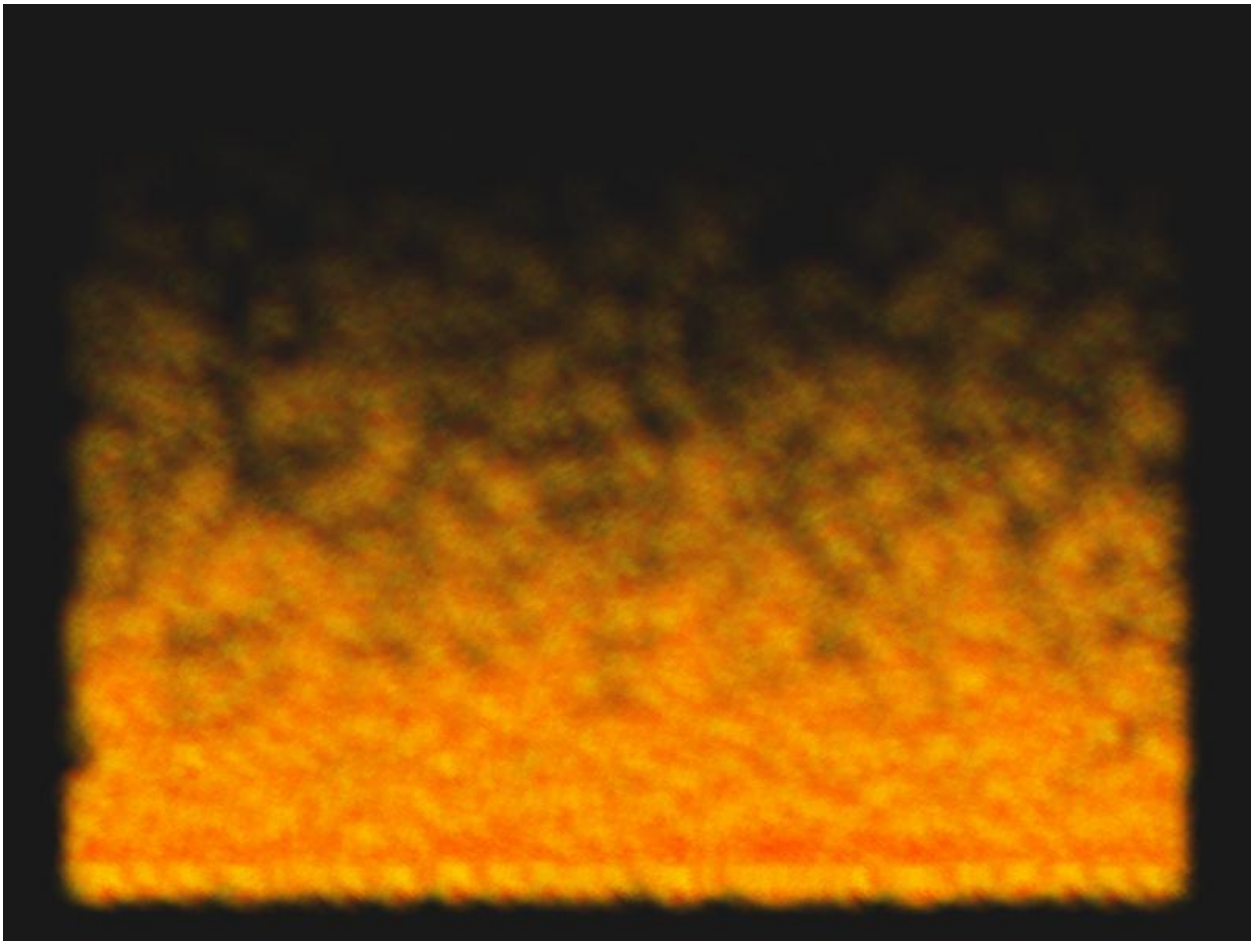
$$P_{n+1} = P_n + \mathbf{v}_n h$$

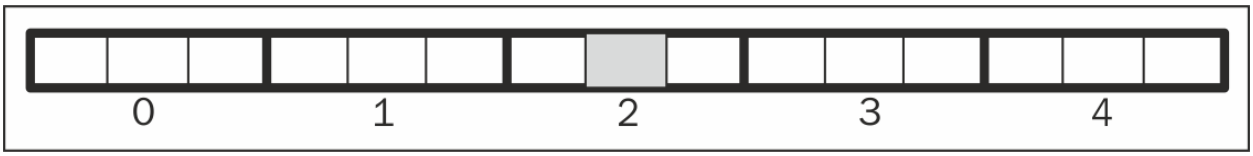
$$\mathbf{v}_{n+1} = \mathbf{v}_n + \mathbf{a}_n h$$

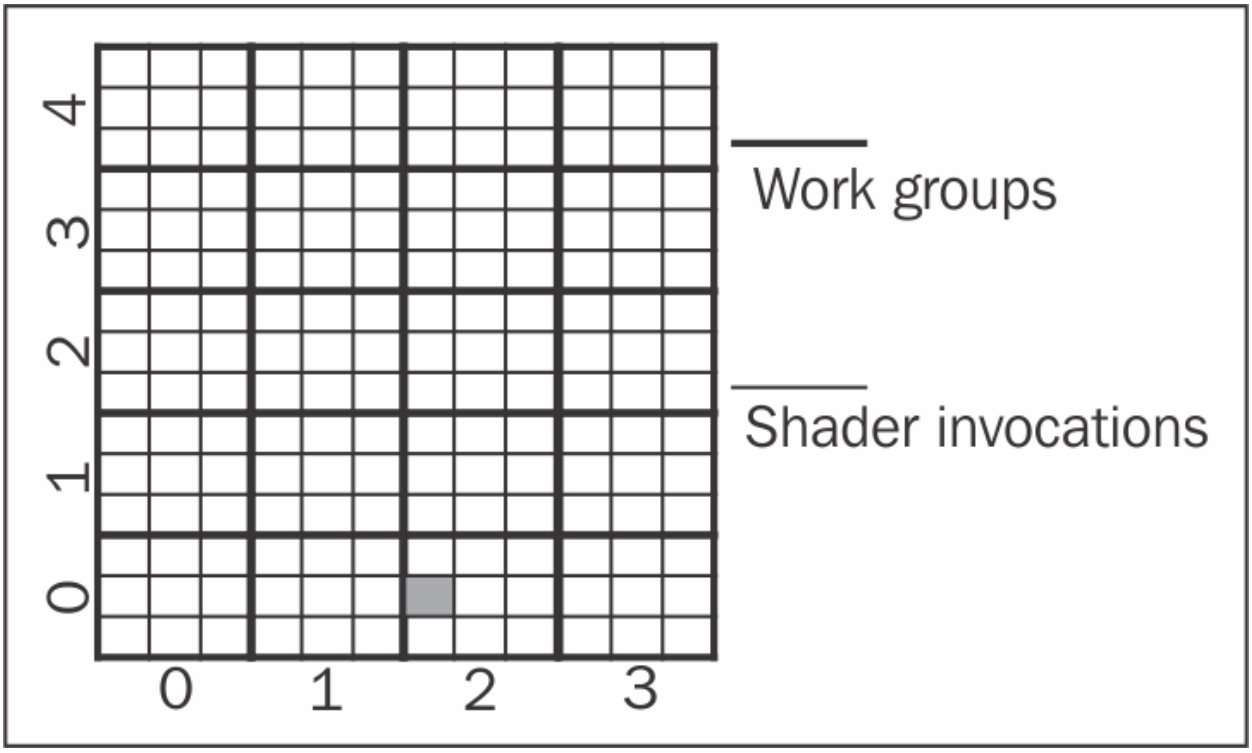


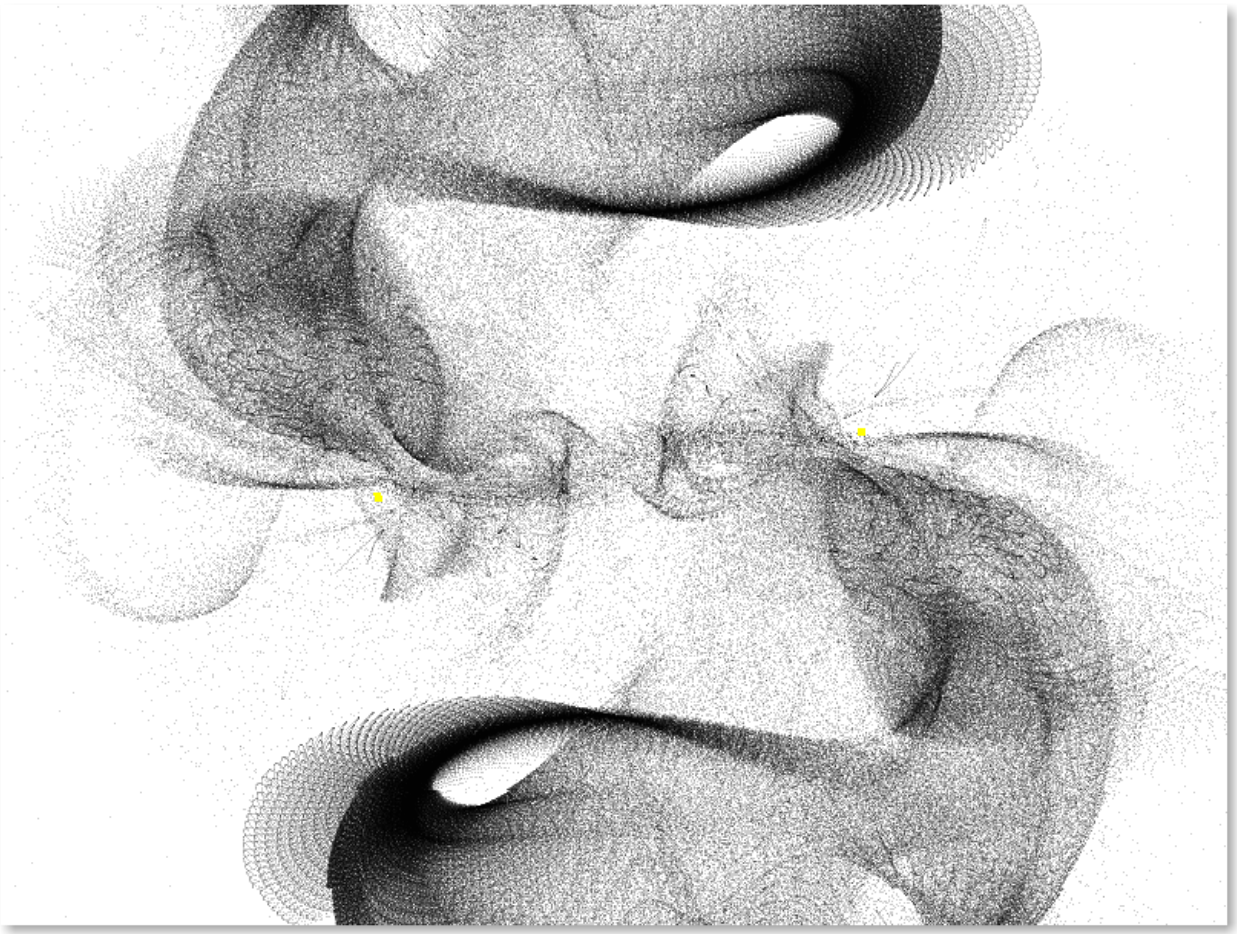

Update pass (no rasterization)
A
input
Vertex Shader
output
B
Render pass
B
input
Vertex Shader
to fragment shader






0
1
2
3
4

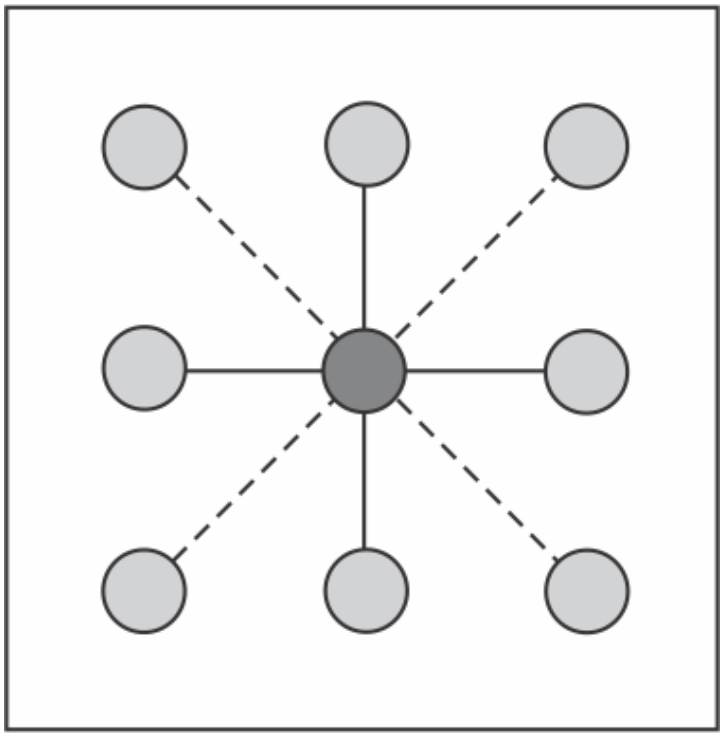

4
3
2
1
0
0
1
2
3
Work groups
Shader invocations



$$\mathbf{F} = \sum_{i=1}^{N} \frac{G_i}{|\mathbf{r}_i|} \frac{\mathbf{r}_i}{|\mathbf{r}_i|}$$

$$\mathbf{P}(t + \Delta t) = \mathbf{P}(t) + \mathbf{v}(t)\Delta t + \frac{1}{2}\mathbf{a}(t)\Delta t^2$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \mathbf{a}(t)\Delta t$$

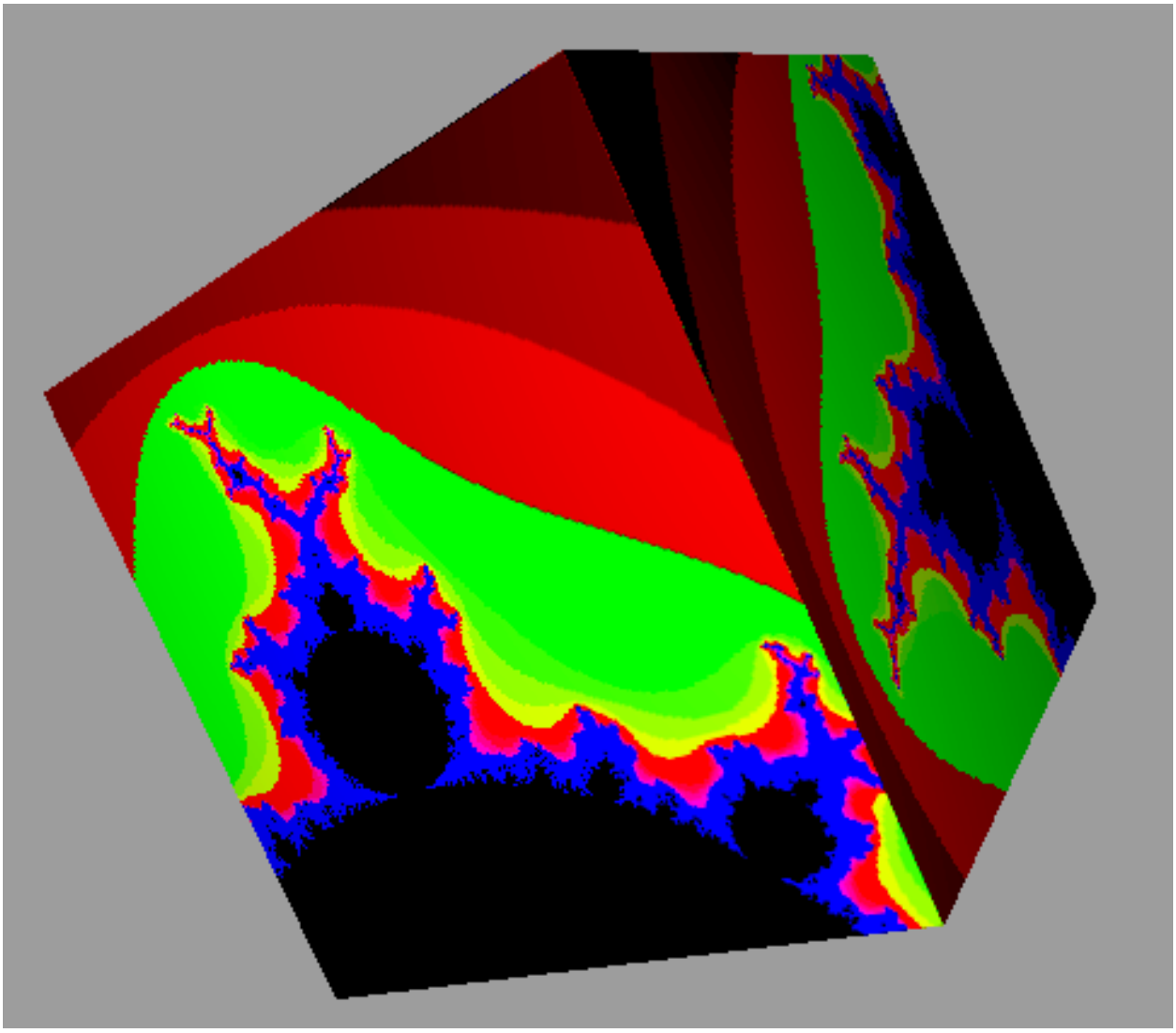


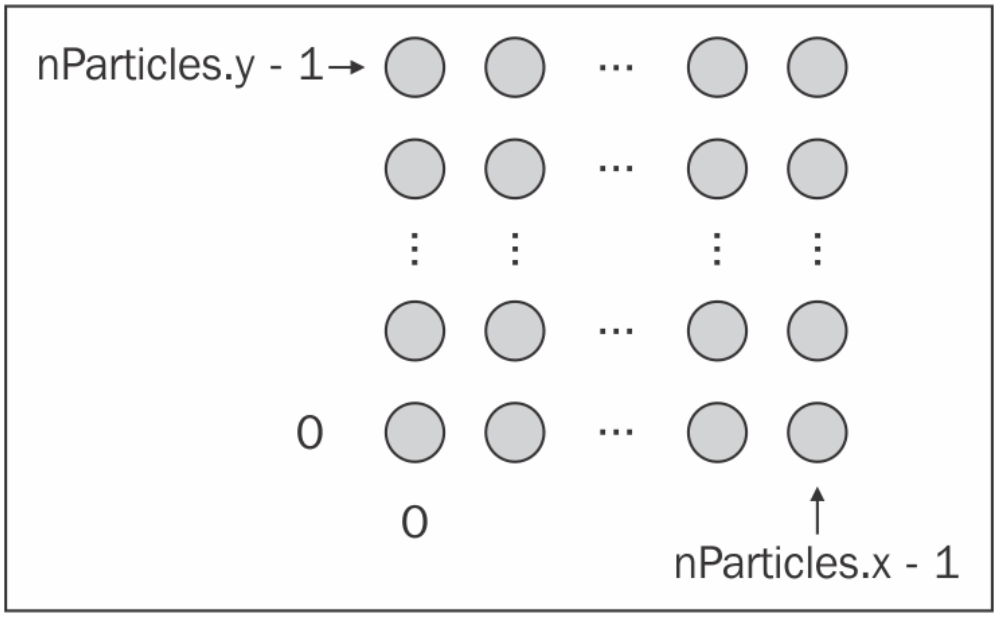
$$\mathbf{F} = K(|\mathbf{r}| - R)\frac{\mathbf{r}}{|\mathbf{r}|}$$

$$\mathbf{F} = -D\mathbf{v}$$

$$\mathbf{S}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \mathbf{S}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

$$z_{n+1} = z_n^2 + c$$




nParticles.y - 1
0
0
nParticles.x - 1