# Production Volume Rendering

## Design and Implementation

Magnus Wrenninge

CRC Press
Taylor & Francis Group
AN A K PETERS BOOK

# Production Volume Rendering

This page intentionally left blank

# Production Volume Rendering

## Design and Implementation

Magnus Wrenninge

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

AN A K PETERS BOOK

Cover design by Vincent Serritella.

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742


Version Date: 20120710

International Standard Book Number-13: 978-1-4398-7363-2 (eBook - PDF)

To Chia-Chi.

This page intentionally left blank

# Contents

# Preface

Production volume rendering, described in less than 15 words, refers to rendering of nonsurfaces for the purpose of creating images for film or animation production. A slightly longer explanation might include some examples, such as the creation and rendering of virtual smoke, fire, dust, and clouds. The "and" in "creation and rendering" is also an important point: production volume rendering refers just as much to the *modeling* of the effects as it refers to the actual *rendering*, and in fact, the modeling side is often the lesser known part of the two.

No matter which way you look at it, production volume rendering is an esoteric subject. Volume rendering in the general sense encompasses everything from medical visualization to real-time game graphics, but production volume rendering has very little overlap with these subjects. There are only a few publicly available pieces of software, and to make matters more complicated, there is also very little research or other documentation available, due to the fact that most development is done in-house at visual effects and animation production houses around the world.

The consequence of this lack of information is that each person who is tasked with working on a production volume renderer, or who just happens to be interested in the topic, has to start out from scratch, inventing most of the puzzle pieces that he or she needs to make their systems work.

A certain frustration with this lack of documentation and systemization led to a 2010 SIGGRAPH course called *Volumetric Methods in Visual Effects*, and the course, in turn, led to the book you now hold in your hand.

The course tried to give an overview of all the different techniques and ideas that are used in the volume-rendering solutions at some of the largest production facilities, and it also covered a wide range of concrete examples of those techniques, as they are implemented at those facilities.

This book takes over where the first part of the course left off and goes into all the same fundamental topics. But where the course stayed in a

generic context, with simple examples, this book provides and describes a systematic implementation of the techniques.

The book's approach is highly pragmatic. Its scope is limited to the techniques and algorithms that are actively used in production work. It leaves out much of the available research into photorealistic volume rendering, but the reason for doing so is simply that those techniques rarely get used in production work.

In order to ensure that enough detail is provided on how each technique works, the book is written around an open source renderer called PVR, which can be downloaded, compiled, and modified by the reader. This approach hopefully means that whatever questions are left by the book can be answered by looking at how the code is implemented.

## Goals

The goal of the book is to provide two paths towards understanding production volume rendering. On one side, it describes the techniques used for modern production volume rendering in a generic context. It shows how the techniques fit together, and how the modules that make it up are used to achieve real-world goals. But it wouldn't be a complete book if it did not also describe an implementation of those techniques. Showing how to translate the abstract set of concepts into concrete, working code is an important part of the equation. It shows that the ideas work. And it shows that they work together to create a complete system.

Throughout this book, the illustrations and rendered images are all created using the code that the book describes. In fact, the scripts and the data used for all the examples are freely available (along with the source code) at http://www.github.com/pvrbook.

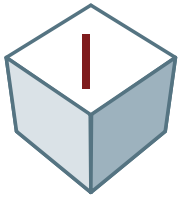
The aim with this approach is to let the readers explore the book in two ways: Someone who is curious about how production volume rendering works in the big picture can start with reading the chapters that describe the fundamental ideas at play. At the same time, if the reader comes across an interesting image, he or she can go straight to the lowest level, starting with the script that created it. The reader can then trace his or her way back through the rendering code and the modeling primitives, which then hopefully illustrates the bigger picture that ties all the steps together.

The primary goal of the book is to be illustrative, and an important step towards that goal is making sure that the translation of the techniques and concepts into the working code is clear and that the code's modular structure properly reflects the ideas presented in the book. Secondly, the concepts outlined in the book and implemented in the code

must be completely integrated, in the sense that each part of the system should work with all the others, using the same fundamental concepts. There should not be any isolated ideas that only work on their own and cannot be integrated into the system.

Efficiency is not the primary concern of the system. In the choice between simpler code or faster performance, PVR strives to be easy to understand. That is not to say that the system's performance is ignored. On the contrary, scalability is a very important aspect of any system that is to be used in production. The goal of the system is instead to show how various design considerations have an impact on both scalability, extensibility, generality, and performance, but to do so in a straightforward and understandable way.

This page intentionally left blank



# The PVR System

The system described in this book is called PVR, standing simply for production volume rendering. Because production volume rendering is a fairly esoteric topic, it was clear early on that a working example of the algorithms and techniques in action had to be included. To not include one would be to rob the reader of the understanding that can sometimes only be had by seeing how a set of techniques covering different areas work in conjunction with one another.

## 1.1 C++ and Python

PVR is implemented entirely in C++. The choice of C++ was simple; although the PVR renderer isn't built primarily for speed, the amounts of data it needs to process makes C or C++ the only real choice. And because the system needed to be modularized so that concepts from the book could map cleanly to code, the object-oriented nature of C++ made it the best candidate.

While C++ has many strengths, runtime configurability isn't necessarily one of them. In order to make it easy for the reader to create scene descriptions, PVR also includes a Python module that wraps the entire code base (or at least the relevant 99% of it) so that scenes can be created and rendered with a few lines of Python code.

To exclude the Python interface (or some other scripting language bindings) would mean that the reader would have to compile a full C++ application in order to create a scene. We assume that the reader takes

no pleasure in decoding GCC's template error messages and, therefore, provide the bindings as a convenient, but powerful, scene-modeling and rendering language.

The intent of this book is to describe a simple but complete volume-modeling and rendering system. The focus is on illustrating the most important concepts as clearly as possible, and on describing a modular system where parts can easily be replaced to show the impact of different techniques and system design decisions. Performance, although very important in a production renderer, is not the primary goal of PVR. The system doesn't use any hard-to-understand or obfuscating optimization techniques, but where algorithmic optimizations are possible, those are discussed and implemented.

The system does use some programming concepts that are advanced in nature, but an effort has been made to keep this in the supporting code. The key parts of the system have all been designed to be as simple and instructive as possible.

### 1.1.1 Use of Namespaces

Namespaces are an important part of writing production code. If we consider some common classes one might write for a render, **`Ray`**, **`Vector`**, **`Curve`**, **`Polygon`**, etc., it is not unlikely that one of the other libraries that our system uses might have similarly named classes. If this occurs, the compiler will throw up its hands in confusion and tell you that you already created a **`Vector`** class, and it is not impressed that you are trying to do so again.

C++ provides a concept called *namespaces* that addresses this problem by changing the internal (C++ calls this *mangled*) name of a symbol to include the name of the namespace. This way, the compiler will know the difference between **`Imath::Vec3<T>`** and your **`MyLibrary::Vec3<T>`**. For this reason, all of the C++ source code is wrapped in a **`pvr`** namespace.

PVR also uses namespaces to divide its classes into broad modules, although more for organizational purposes than to prevent symbol name clashes. Representations of geometry, such as polygons, particles, and attributes, are all in the **`pvr::Geo`** namespace. Modeling-related classes are in **`pvr::Model`** and so on.

| | |
|---|---|
| **`pvr::Accel`** | Acceleration data structures |
| **`pvr::Geo`** | Geometry-related classes and functions |
| **`pvr::Math`** | Math-related functions |
| **`pvr::Model`** | Volume-modeling classes |

| | |
|---|---|
| **pvr::Noise** | Noise and fractals |
| **pvr::Render** | Rendering-related classes |
| **pvr::Sys** | System-related classes |
| **pvr::Util** | Various utility functions and classes |

### 1.1.2 Use of External Libraries

Where possible, existing libraries have been used to accomplish tasks not central to the functionality of the system and for components not directly related to volume modeling and rendering.

| | |
|---|---|
| **OpenEXR/Imath** | Used as the main math library. Contains basic classes such as vectors and matrices. |
| **OpenImageIO** | A library for image input and output. Gives PVR the ability to read and write a variety of image formats. |
| **Field3D** | Provides voxel data structures and routines for storing voxel data on disk. |
| **GPD** | A library that enables reading and writing of Houdini's **geo** and **bgeo** formats. |
| **boost** | After the Standard Template Library, probably the most commonly used library in the world. PVR uses a variety of classes from **boost**, from smart pointer classes through timing utilities to Boost.Python. |

## 1.2 Python Bindings

The use of Python in production pipelines has exploded over the last five to ten years. It is most commonly used for general scripting, but due to the ease of creating bindings for existing C/C++ libraries, it has also replaced the internal scripting languages of several major applications, for example, hscript in Houdini, MEL in Maya, and Tcl in Nuke.

PVR uses Boost.Python for its bindings, which exposes PVR's internal classes and function directly. Boost.Python makes it very simple to build the bindings and supports advanced memory management and object lifetime techniques, such as smart pointers, deep and shallow copying, etc.

As an example, the following code snippet builds a Python class that exposes the **`Renderer`** class, automatically manages its lifetime using **`Renderer::Ptr`**, and also replaces Python's default constructor with a custom one.

**Code 1.1.** Creating Python bindings for the **`Renderer`** class using Boost.Python

```
class_<Renderer, Renderer::Ptr>("Renderer", no_init)
    .def("__init__",      make_constructor(Renderer::create))
    .def("clone",         &Renderer::clone)
    .def("setCamera",     &Renderer::setCamera)
    .def("setRaymarcher", &Renderer::setRaymarcher)
    .def("addVolume",     &Renderer::addVolume)
    .def("addLight",      &Renderer::addLight)
    .def("execute",       &Renderer::execute)
    .def("saveImage",     &Renderer::saveImage)
    ...
    ;
```

Boost.Python also makes it easy to expand the flexibility of the C++ code when creating bindings. For example, Python has no concept of **`const`** objects. To make the C++ code play nice with the Python bindings, Boost.Python provides a function that instructs the binding layer that some extra type conversions are legal, for example converting a non-**`const`** object to a **`const`** one.

**Code 1.2.** Instructing Boost.Python that a non-**`const`** pointer is convertible to a **`const`** pointer

```
implicitly_convertible<Renderer::Ptr, Renderer::CPtr>();
```

Boost.Python also makes it simple to adapt C++'s rigid type structure to work within Python's loosely typed classes. For example, we can tell it to allow conversions from Python's **`list`** and **`dict`** types to C++'s **`std::vector`** and **`std::map`**.

**Code 1.3.** Converting a Python type to a C++ type

```
template <typename T>
std::vector<T> pyListValues(boost::python::list l)
{
  using namespace boost::python;
  using namespace std;

  vector<T> hits;

  for (boost::python::ssize_t i = 0, end = len(l); i < end; ++i) {
    object o = l.pop();
    extract<T> s(o);
```

```
    if (s.check()) {
      hits.push_back(s());
    }
  }

  return hits;
}

// Convert a python list, grabbing only float values
boost::python::list l = some_list();
std::vector<float> floatVec = pyListValues<float>(l);
```

The bindings for PVR reside in the `libpvr/python` folder.

## 1.3 Rendering with PVR

To give an example of how rendering is accomplished with PVR, we first look at an example of a Python script that uses the **`Modeler`** class to build a voxel buffer and then renders it using the **`Renderer`** class. In the example, references have been made to each of the relevant chapters and sections. The final image is shown in Figure 1.1.

PVR's Modeling Pipeline, 91
Voxel Buffers, 33
PVR's Rendering Pipeline, 203

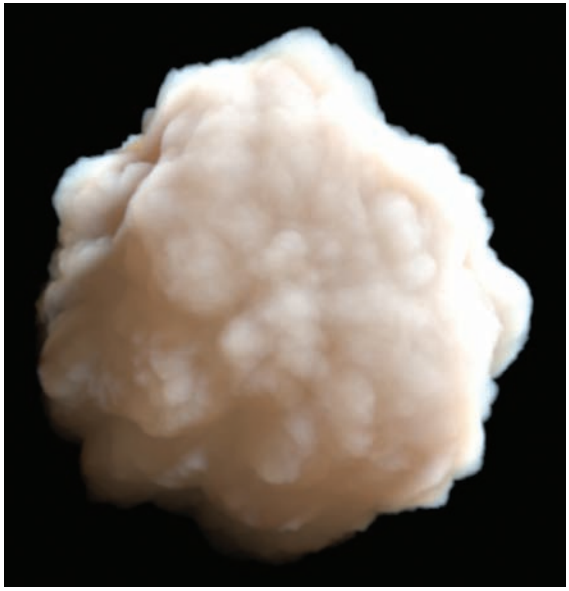


**Figure 1.1.** The result of our PVR example.

First, the **pvr** module is imported into the script. The **from pvr import *** syntax is used instead of **import pvr** so that the symbols in the library become visible without always having to prefix them with the library name.

**Code 1.4.** A simple modeling and rendering example

```
#! /usr/bin/env python

from pvr import *
```

The next step is to create instances of the classes that are used in the volume-modeling process. The **Modeler** is responsible for taking inputs in the form of **ModelerInput** instances and turning them into voxel buffers. Each modeler input contains the definition of a *volume primitive* (also known as *volumetric primitive*), in this case, a **PyroclasticPoint**, and a **Geometry** instance, which is the underlying geometric representation of the volume primitive. Pyroclastic points are a type of rasterization primitive, and their geometric representation is a **Particles** instance.

Fundamentals of Volume Modeling, 75
**Modeler**, 92
**ModelerInput**, 97
**Rast::PyroclasticPoint**, 122
**Geometry**, 20
Rasterization Primitives, 79
Using Geometry to Guide Volumetric Primitives, 82
**Particles**, 21

**Code 1.5.** A simple modeling and rendering example

```
# Modeling classes
modeler = Modeler()
parts   = Particles()
geo     = Geometry()
prim    = Prim.Rast.PyroclasticPoint()
input   = ModelerInput()
```

The particles object is configured with a single point, which by default sits at the origin. The particles instance is then hooked up to the geometry container, and the two are added to the modeler input along with the volume primitive. In a normal render, the volumetric primitive would be configured with a set of *attributes* to drive its appearance. In this simple example the default parameters will be used.

Geometry Attributes, 24

**Code 1.6.** A simple modeling and rendering example

```
# Create a modeling primitive with a single input point
parts.add(1)
geo.setParticles(parts)
input.setGeometry(geo)
input.setVolumePrimitive(prim)
```

The final step in the volume-modeling process is to add the **ModelerInput** to the **Modeler** and then update the *bounds*, which configure the

**ModelerInput**, 97
**Modeler**, 92
Defining the Voxel Buffer, 76

voxel buffer so that it encloses all of the primitives in the modeler's list of inputs. Once the final resolution of the voxel buffer has been set, the modeler is executed, which runs each of its inputs and lets them write their data to the voxel buffer.

**Code 1.7.** A simple modeling and rendering example

```
# Add input to modeler and rasterize
modeler.addInput(input)
modeler.updateBounds()
modeler.setResolution(200)
modeler.execute()
```

**Renderer**, 204

Volumetric Lighting, 167
**PerspectiveCamera**, 220
Cameras, 18
Raymarching in PVR, 267
**Raymarcher**, 268

**RaymarchSampler**, 285

**Light**, 292

**Occluder**, 311

**VoxelVolume**, 243

Next, we create the rendering-related objects. The **Renderer** is the most important one, responsible for firing the rays that are turned into the pixel values of the final image. Any render requires that a camera be present; in this example, a **PerspectiveCamera** is used.

PVR uses raymarching to integrate the volumetric properties in the scene, but the task is broken into two separate parts. The **Raymarcher** takes steps along each ray fired from the final image's pixels, but the radiance and transmittance change at each step is determined by a **RaymarchSampler**.

Various types of light sources may be used in PVR to produce illumination of the scene. The task is divided between two classes. Each **Light** determines the illumination intensity and direction present at various points in the scene, but answering queries about how much light actually arrives at any given point is done by the **Occluder** rather than the light itself.

Although the modeler produces a voxel buffer after the **execute()** method has been called, it is not in a form that the renderer can handle. Instead, we first wrap it in a **VoxelVolume** instance, which makes the buffer renderable.

**Code 1.8.** A simple modeling and rendering example

```
# Rendering classes
renderer        = Renderer()
camera          = PerspectiveCamera()
raymarchSampler = PhysicalSampler()
raymarcher      = UniformRaymarcher()
occluderCamera  = SphericalCamera()
occluder        = OtfTransmittanceMapOccluder(renderer, occluderCamera, 8)
light           = PointLight()
volume          = VoxelVolume()

# Configure rendering objects
camera.setPosition(V3f(0.0, 0.25, 6.0))
camera.setResolution(V2i(320, 240))
```

```
raymarcher.setRaymarchSampler(raymarchSampler)
light.setPosition(V3f(10.0, 10.0, 10.0))
light.setIntensity(Color(1.5))
occluderCamera.setPosition(light.position())
light.setOccluder(occluder)
volume.addAttribute("scattering", V3f(4.0, 6.0, 8.0))
volume.setBuffer(modeler.buffer())
```

Before the final image can be rendered, the camera, raymarcher, volume, and light all need to be added to the **`Renderer`**. Taken together, the **`Volume`** and **`Light`** instances make up the **`Scene`** of the render.

The last step before executing the render is to print the scene information. The **`execute()`** call will then start firing rays through the scene, which ultimately results in an image that can be saved to disk.

**`Renderer`**, 204
**`Volume`**, 233
**`Light`**, 292
**`Scene`**, 203

**Code 1.9.** A simple modeling and rendering example

```
# Connect the renderer with the raymarcher, camera, volume and light
renderer.setRaymarcher(raymarcher)
renderer.setCamera(camera)
renderer.addVolume(volume)
renderer.addLight(light)

# Print scene structure and start render
renderer.printSceneInfo()
renderer.execute()

# Save image to disk
renderer.saveImage("out/image.png")
```

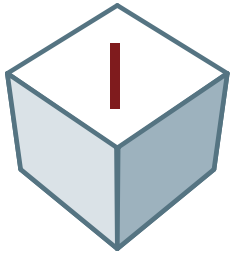
PVR reports progress and status information at each step in the pipeline. The example code above generates the following log output, which gives feedback on each of the steps in the process.

**Code 1.10.** The log file generated by the above rendering example

```
23:01:33 [pvr] Updated bounds to: ((-3 -3 -3), (3 3 3))
23:01:33 [pvr] Creating dense buffer
23:01:33 [pvr] Using uniform/matrix mapping
23:01:33 [pvr] Setting voxel buffer resolution to: (200 200 200)
23:01:33 [pvr] Pyroclastic point primitive processing 1 input points
23:01:35 [pvr]   Rasterization: 22.68%
23:01:38 [pvr]   Rasterization: 45.09%
23:01:40 [pvr]   Rasterization: 67.70%
23:01:43 [pvr]   Rasterization: 90.19%
23:01:44 [pvr]   Time elapsed: 11.2419996
23:01:44 [pvr] Voxel buffer memory use: 91MB
23:01:44 [pvr] Scene info:
23:01:44 [pvr]   (VoxelVolume)
23:01:44 [pvr]     a scattering
```

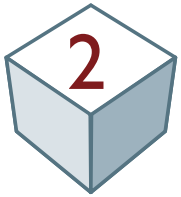
```
23:01:44 [pvr]     i scattering : (4 6 8)
23:01:44 [pvr]     i Empty space optimization disabled
23:01:44 [pvr]     p Isotropic
23:01:44 [pvr]   (PointLight)
23:01:44 [pvr]     i (1.5 1.5 1.5)
23:01:44 [pvr]     o OtfTransmittanceMapOccluder
23:01:44 [pvr] Rendering image (320 240) (1 x 1)
23:01:46 [pvr]   44.23%
23:01:49 [pvr]   88.77%
23:01:49 [pvr]   Time elapsed: 5.51499987
23:01:49 [pvr] Writing image: out/image.jpg
23:01:49 [pvr]   Done.
```

This page intentionally left blank




# Fundamentals

This page intentionally left blank




# The Basics

## 2.1 Time and Motion Blur

One of the key requirements of a production renderer is that it must handle the effects of time-varying properties robustly. Because a camera shutter (see Figure 2.1) stays open over a finite amount of time, we must account for changes in the scene during that time period. Attributes that often change with time are object position (transformation motion blur), object shape (deformation motion blur), and camera position and orientation (camera motion blur). For a rendering solution to be useful in a production context, each of these must be handled.

Time itself can have multiple different reference frames, each of which can be useful in a renderer. Film and television rely on showing a sequence of images that change quickly enough to give the illusion of motion. Each of these images is referred to as a *frame*, and an integer number can be assigned to each frame, making for a convenient reference frame for time.

Time can, of course, also be measured in seconds. If we consider an animation containing 48 frames, running at 24 frames per second, we have two seconds of material, given that the *frame rate* is the standard 24 frames per second (often abbreviated fps). If we start numbering frames at 1, we can convert between time ($t$) and frame number ($F$) using a simple formula:

$$t = \frac{F - 1}{\text{frames per second}}.$$

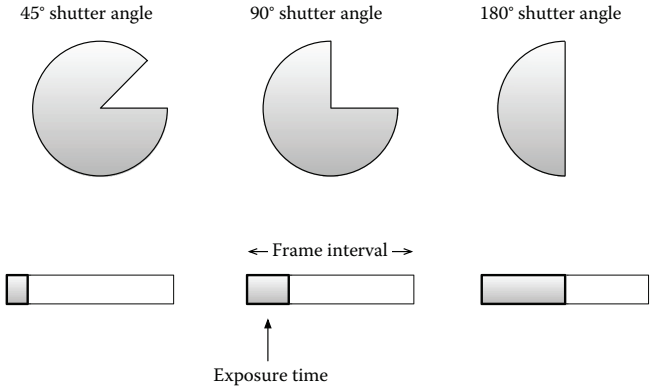



**Figure 2.1.** A motion picture camera shutter uses a rotating disk as its shutter. By varying the size of the open section, a certain shutter angle and exposure time is achieved.

A third way of looking at time is to define it in the time interval bounded by the camera shutter opening and closing. Here we can consider a parametric measure of time that is unitless and is defined only within a $[0, 1]$ interval.

The shutter is usually not open for the entire duration of a frame. Most motion pictures are filmed using a *shutter angle* of 180° [Burum 07], a notion that warrants some explaining. A motion picture camera's shutter is most often a rotating disk with an adjustable slot, which in turn controls how long the exposure of each film frame is. The degree measure refers to the angle the slot occupies on the disk, meaning that at 180° the shutter is open 50% of the time and is closed 50% of the time. In computer graphics, this is more often expressed as a fraction called *shutter length* or *motion blur length*, and the corresponding value to 180° is 0.5.

We often refer to the actual time that the camera shutter is open as $dt$, and we can find it using the frame rate and shutter angle or motion blur length:

$$dt = \frac{\text{motion blur length}}{\text{frames per second}} = \frac{\text{shutter angle}/360^\circ}{\text{frames per second}}.$$

A strict and consistent definition and handling of the various time frames is important in a renderer, and in PVR, they are actual class types, which prevents accidental misinterpretation. Time in seconds is defined by the **`Time`** class, whereas shutter open/close time is defined by **`PTime`**. PVR does not deal with frame time directly.

**Code 2.1.** The `Time` class

```
class Time
{
public:
  explicit Time(const float t)
    : m_value(t)
  { }
  operator float() const
  { return m_value; }
  float value() const
  { return m_value; }
private:
  float m_value;
};
```

**Code 2.2.** The `PTime` class

```
class PTime
{
public:
  explicit PTime(const float t)
    : m_value(t)
  { }
  operator float() const
  { return m_value; }
  float value() const
  { return m_value; }
private:
  float m_value;
};
```

Geometry Attributes, 24

In PVR, the motion of particles and geometry is represented using the $v$ (for *velocity*) point attribute (see Section 2.4). Velocity vectors are always defined in m/s, meters per second. *Motion vectors* are different from velocity vectors; they refer to motion in the current shutter open/close interval and thus only have the length unit $m$. Any velocity vector can be converted to a motion vector through the formula

$$\vec{m} = \vec{v} \cdot dt.$$

### 2.1.1 Render Globals

When designing a renderer, there is always a certain amount of information that pertains to all parts of the renderer. Rather than passing a set of variables to every single rendering function, it is quite common to use an

object that is globally accessible to store such information. In PVR, this class is called **RenderGlobals**.

The **RenderGlobals** class stores pointers to the current scene as well as to the render camera. Information relating to time is also available in the class. When a new render frame is initiated, the rendering class configures **RenderGlobals** by calling the static **setupMotionBlur()** method. After that, any of PVR's components can access *dt* through the **RenderGlobals::dt()** method.

**Code 2.3.** **RenderGlobals**

```
class RenderGlobals
{
public:
  // Typedefs
  typedef boost::shared_ptr<const pvr::Render::Scene> SceneCPtr;
  typedef boost::shared_ptr<const pvr::Render::Camera> CameraCPtr;
  // Exceptions
  DECLARE_PVR_RT_EXC(BadFpsException, "Bad frames per second value:");
  DECLARE_PVR_RT_EXC(BadShutterException, "Bad shutter value:");
  // Main methods
  static void       setupMotionBlur(const float fps, const float shutter);
  static void       setScene(SceneCPtr scene);
  static void       setCamera(CameraCPtr camera);
  // Accessors
  static float      fps();
  static float      shutter();
  static float      dt();
  static SceneCPtr  scene();
  static CameraCPtr camera();
private:
  // Data members
  static float      ms_fps;
  static float      ms_shutter;
  static float      ms_dt;
  static SceneCPtr  ms_scene;
  static CameraCPtr ms_camera;
};
```

### 2.1.2 Function Curves

So far, we have mentioned motion blur, but we have not yet showed how time-varying properties can be represented. PVR treats properties that change with time as one-dimensional functions that are linearly *interpolated* to find sub-sample values. (See, for example, Figure 2.2.)

In most cases, time-varying properties change gradually and smoothly over time and can be described well using only two time samples: one at the shutter open time and one at shutter close. In other cases, such as

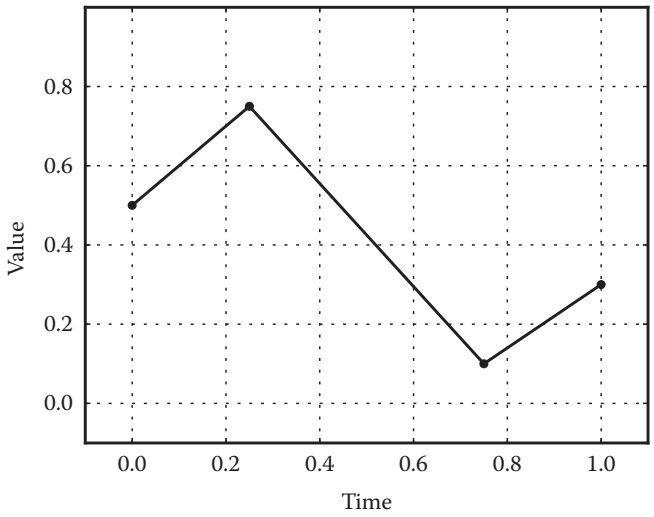



**Figure 2.2.** A function curve using four samples and linear interpolation.

a camera with "shake" applied, the changes can be drastic and sudden, requiring more than two samples to accurately describe the change in position, orientation, etc. In order to support arbitrary intra-frame motion, PVR handles these time-varying properties using the **`Util::Curve`** class. Each **`Curve`** can contain as many samples as is necessary, and in-between values are then linearly interpolated.

**Code 2.4.** **`Curve`**

```
class Curve
{
public:
...
  // Main methods
  void               addSample(const float t, const T &value);
  T                  interpolate(const float t) const;
  size_t             numSamples() const;
  const SampleVec&   samples() const;
  std::vector<float> samplePoints() const;
  std::vector<T>     sampleValues() const;
  void               removeDuplicates();
  static CPtr        average(const std::vector<CPtr> &curves);
private:
...};
```

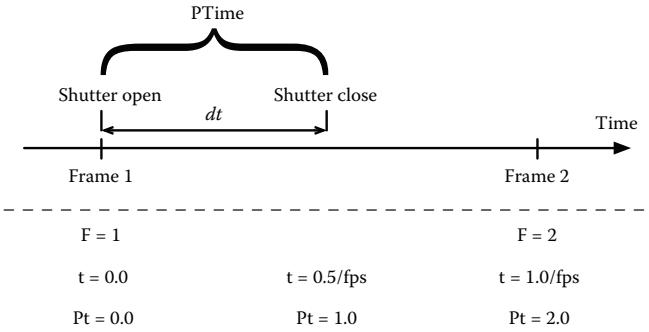



**Figure 2.3.** PVR's temporal coordinate frames.

The **Curve** class is templated and can theoretically be used to store any type of data that can be interpolated. The most commonly used types, however, are

```
typedef Curve<float>  FloatCurve;
typedef Curve<Color>  ColorCurve;
typedef Curve<Vector> VectorCurve;
typedef Curve<Quat>   QuatCurve;
```

When a **Curve** is used in PVR, the time dimension is assumed to line up with **PTime**, such that $t = 0.0$ refers to the start of the frame and the shutter open time. The end of the shutter interval falls at $t = 1.0$, which matches the **PTime** definition. When constructing a curve, it is most common to have information about how a given attribute changes from frame to frame. If we let $t_P^i = 0.0$ be the **PTime** of the current frame start, then the equivalent **PTime** for the next frame start can be found through the expression

**PTime**, 15

$$t_P^{i+1} = \frac{1}{\text{motion blur length}}.$$

Figure 2.3 illustrates PVR's definition of time reference frames for a motion blur length of 0.5.

## 2.2 Cameras

The camera classes in PVR are very simple, and their implementations should provide no surprises to the reader. Most parameters are assumed

to be temporally varying, meaning that properties such as position, orientation, and field of view may change over the course of the current frame. In support of this, all of the calls to the camera that depend on time use the **`PTime`** concept, where 0.0 is assumed to be the start of the current frame, i.e., when the shutter opens, and 1.0 is the time that the shutter closes.

This book assumes the reader is familiar with common computer graphics conventions for camera projection calculations. For a thorough introduction to cameras in computer graphics, [Watt 00] and [Pharr and Humphreys 10] are good sources.

### 2.2.1 Camera Coordinate Spaces

PVR's world space is right-handed, but its camera space is left-handed. By default, a camera with no rotation looks down the negative $z$-axis in world space, and down positive $z$ in camera space. The camera's $x$- and $y$-axes line up with the world space, with $x$ to the right and $y$ facing up. (See Figure 2.4.)

Screen space defines the projected view of the camera, with $x = \quad 1$ indicating the left edge of the camera's view, $x = 1$ the right edge, $y = \quad 1$ the bottom edge, and $y = 1$ the top edge. The depth dimension has no negative range, instead putting $z = 0$ at the near plane of the projection and $z = 1$ at the far plane. (See Figure 2.5.)

PVR also uses a second projection space, which is called NDC space. NDC stands for normalized device coordinates and changes the $x$ and $y$ ranges of the projection to the $[0, 1]$ range. (See Figure 2.6.)

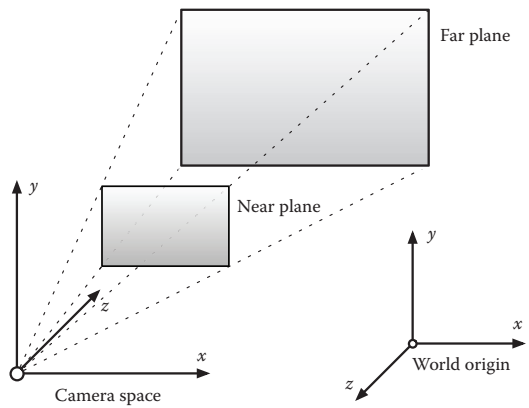



**Figure 2.4.** PVR's camera space conventions.