Multiflash Lens

Introduction to custom RixProjection plug-ins

Patrik S. Hadorn

October, 2016

This tutorial covers the basic principles you'll require to write your own *RixProjection* plug-ins, also known as *Lens Shader*s. These plug-ins give a developer full control over how a camera "sees" the scene. Not only does this allow the implementation of custom camera projections, it also opens up a lot of new exciting possibilities which you don't get out of the box.

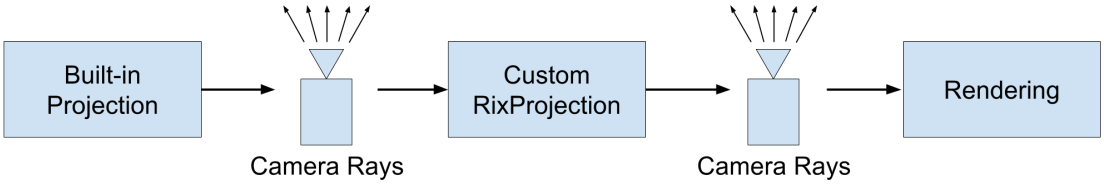
Customization of the camera is something which hasn't always been possible in *RenderMan*. There were tricks which used the raytracing engine for these kind of effects but they never compared to the performance and flexibility now available in *RIS* with its *RixProjection* interface.

This lesson is tailored for people who are new to the RIS API and it requires a minimal

amount of C/C++ knowledge. I hope to help bridging that gap between being artistic and being technical, showing that these skills can go hand-in-hand. *RenderMan* is an extremely powerful and flexible renderer and if you know how to leverage that, you'll really be able to achieve that look you're after!

1 Introduction to RixProjections

Projection plug-ins are essentially modifiers applied to a bunch of camera rays before they're being traced through the scene. It's possible to adjust just individual aspects of these rays while keeping everything else untouched from the built-in projection (being *perspective* or *orthographic*). This is, for example, how the *PxrRollingShutter* plug-in affects motion blur while leaving everything else, be it field-of-view (*FOV*) or depth-of-field (*DOF*), untouched.



*RixProjections modify the camera rays generated by the built-in (*orthographic *or* perspective*) projection.*

1.1 Simple example

Let's jump right into our first example. Don't worry about what the code means right now, this is just to have something we can work with for now:

```
1 #include <RixInterfaces.h>
2 #include <RixProjection.h>
3
4 class SimpleProjection : public RixProjection
5 {
6 public:
7   // Don't forget to make your destructor virtual or it might not get
  ↪  called!
8   virtual ~SimpleProjection(){}
9
10  virtual int Init(RixContext& ctx, char const* pluginPath)
11  {
12    // Get the RixMessages interface so we can print a message to the log
```

```
13      RixMessages* msgs =
    ↪  static_cast<RixMessages*>(ctx.GetRixInterface(k_RixMessages));
14      msgs->InfoAlways("SimpleProjection::Init()");
15
16      // Returning '0' means that there was no error. Anything else indicates
    ↪  an error.
17      return 0;
18    }
19
20    virtual void Finalize(RixContext& ctx) {}
21
22    virtual RixSCParamInfo const* GetParamTable()
23    {
24      // RixSCParamInfo() signals the end of the parameter list to PRMan and
    ↪  has to be included even if we don't want any parameters.
25      static RixSCParamInfo params[] = {RixSCParamInfo()};
26      return params;
27    }
28
29    virtual void RenderBegin(
30        RixContext& ctx,
31        RixProjectionEnvironment& env,
32        RixParameterList const* params) {}
33
34    virtual void RenderEnd(RixContext& ctx) {}
35
36    virtual void Synchronize(
37        RixContext& ctx,
38        RixSCSyncMsg syncMsg,
39        RixParameterList const* params) {}
40
41    virtual void Project(RixProjectionContext& pCtx) {}
42
43  private:
44  };
45
46  // RIX_PROJECTIONCREATE is a macro for the CreateRixProjection function
47  RIX_PROJECTIONCREATE
48  {
49    return new SimpleProjection;
50  }
51
52  // RIX_PROJECTIONDESTROY is a macro for the DestroyRixProjection function
53  RIX_PROJECTIONDESTROY
54  {
55    delete static_cast<SimpleProjection*>(projection);
56  }
```

Listing 1: SimpleProjection.cpp

This is pretty much the smallest projection plug-in possible. It doesn't do any changes on camera rays yet, but it defines all the methods required for *RenderMan* to load and execute it as a custom projection. It's good to keep a copy of this somewhere so you can simply copy-paste it whenever you start writing a new projection plug-in.

There is one interesting thing happening though. If you look a bit closer at the *Init* method you'll see that we're getting a *RixMessages* interface from the *RixContext*. With this, we can print a message to the log, showing that our plug-in is indeed being called by *RenderMan*. The *RixInterfaces* are an integral part of *RenderMan*'s *API* and there are many useful things we can do with it, besides just printing to the log. Have a look at the *RixInterfaces* Documentation when you're ready to learn more about it.

```
12      // Get the RixMessages interface so we can print a message to the log
13      RixMessages* msgs =
    ↪   static_cast<RixMessages*>(ctx.GetRixInterface(k_RixMessages));
14      msgs->InfoAlways("SimpleProjection::Init()");
```

To use this code, you'll have to create a library out of it. With the *GCC* compiler this would be:

```
g++ -o SimpleProjection.so -shared -fPIC -O2
 ↪  -I/opt/pixar/RenderManProServer-21.0/include SimpleProjection.cpp
```

On Windows, you can use Visual Studio to create a library by setting the application type to *DLL*.

Once we've created the library, we'll need a render we can test it with. For this, I suggest to use a simple *RIB*. This allows us to iterate quickly and the complexity of the scene won't disturb from our development. If you don't like working with *RIB* files and prefer using *Maya*, have a look at our lesson on REF[How to load custom plug-ins in *RenderMan for Maya*].

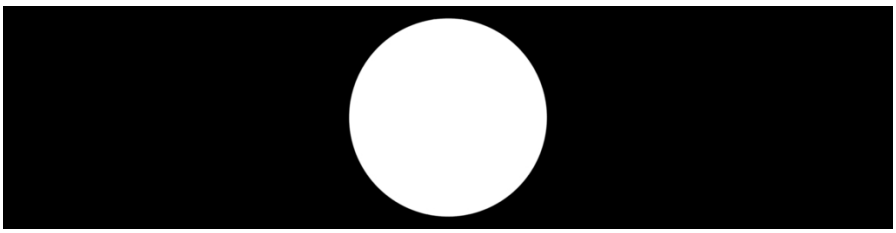
To be able to run *PRMan* as a standalone, you need to have *RenderMan ProServer* installed (also available with a *non-commercial* license), just use your *RenderMan Installer* to install it. Brian Savery has written an introduction on how to set up and use *PRMan* as a standalone: Using *RenderMan* without *Maya*.

```
1 Integrator "PxrPathTracer" "handle"
2
3 Format 1024 256 1
```

```
4 Display "preview" "it" "rgba"
5
6 # This is the builtin projection
7 Projection "orthographic"
8
9 # This loads our custom projection plug-in!
10 Projection "./SimpleProjection" # <-- This loads our custom projection
   ↪  plug-in!
11
12 ScreenWindow -4.5 4.5 -1.125 1.125
13
14 Translate 0 0 4
15 WorldBegin
16   Bxdf "PxrConstant" "constant"
17   Sphere 1 -1 1 360
18 WorldEnd
```

Listing 2: SimpleRIB.rib

This *RIB* can be rendered by typing *'prman SimpleRIB.rib'* in the terminal. If you're getting an error message about 'prman' not being found, make sure you've setup the RMANTREE and PATH environment variable correctly as explained in Brian's article. This should render a white sphere over black background:



Rendering of the example RIB

If our plug-in got loaded and run successfully, you should see the following message in your log:

```
X00004 SimpleProjection::Init()
```

If you're getting a warning message that the plug-in couldn't be loaded, make sure it's in the same directory as your *RIB* file. We've specified a path for our custom plug-in which is relative to where the *RIB* is located:

```
Projection "./SimpleProjection"
```

Congratulations, you've just written and executed your very first projection plug-in! It might not be doing anything fancy yet but we're getting to that, so stay with me!

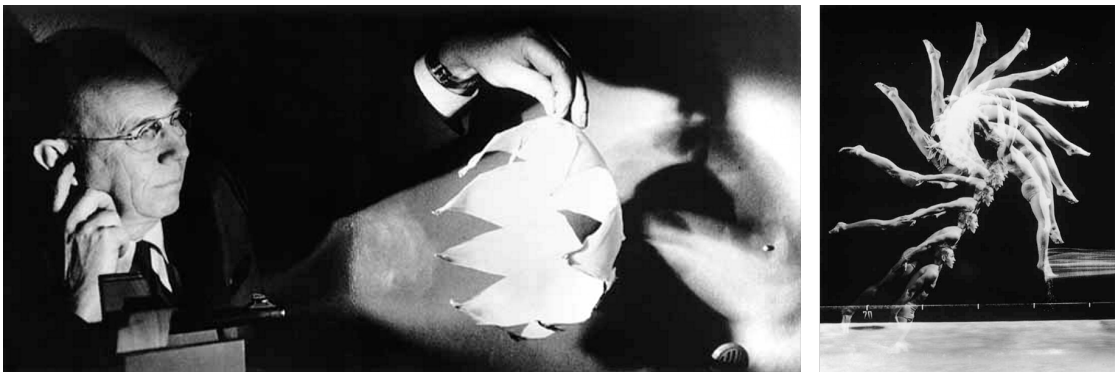
2 The Multiflash Lens

Now that we've covered the basics, we're ready to start with the main project of this lesson: The Multiflash Lens! This plug-in is based on *Multiflash Photography* from traditional photography.

2.1 Multiflash Photography

Multiflash photography is a technique where a camera sensor is being exposed to the scene multiple times, making it related to multiple exposure photography. However, this technique relies on a flash to repeatedly illuminate the scene.

Harold E. Edgerton (1903-1990) has pioneered this discipline by establishing the stroboscope as a tool in photography instead of just being used in laboratories. With this equipment he was able to take pictures with very fast shutter speeds. More interestingly, this allowed him to take multiple of these short exposures in one single image. This reveals motion in an unprecedented and precise way.



left: Self portrait of Harold E. Edgerton and a bursting ballon; right: Multiflash photography of Burt West flipping backwards, ©2010 MIT. Courtesy of MIT Museum.

If we'd want to replicate this effect on a computer, one approach would be to render multiple images at different times and composite them on top of each other. While this is a very easy setup, it has some issues. Having to render several images increases the

time required to create one complete frame. If we want to adjust the intervals between exposures the setup quickly becomes more complex and impractical for larger projects.

If there was a way to render multiple exposures in one image straight out of *RenderMan*, these limitations would be removed. *Adaptive sampling* would make sure we render efficiently, focusing samples where they really matter. Even better, we'd be able to light the scene using re-rendering while looking at the final picture instead of just one frame. The additional control we gain from writing a custom *RixProjection* plug-in would allow us to push the effect further than what common multiflash photography can do.

2.2 Time sampling

A direct translation of multiflash photography would be to animate the lights in your scene to flash for very short amounts of time. When rendering with motion blur, this would then create the stroboscopic effect we're after. However, due to the way a path tracer samples time, this would result in very noisy results.

To render motion blur, a path tracer shoots rays for specific times. For example, when rendering frame 1 of your animation, *RenderMan* would shoot rays, uniformly distributed between this frame and the next (or even the one before when your shutter is centered). If an object is only visible for a fraction of a frame, many rays have to be shot to get a clean result. This is why very fast moving or sparkling objects can take a long time to render cleanly. Here's an example of uniform time sampling:

A moving sphere rendered with uniform time sampling

Since flashing lights won't render well, we'll approach it from a multiple exposure point of view. By changing the time samples for each ray, we can adjust how *RenderMan* renders motion blur. We could, for example, reset the time for each ray to 0 which essentially disables motion blur. But this could as well be interpreted as a single flash at the beginning of the frame. What would be more interesting is to set the time of half the rays to 0 and the other half to 1, which would already give us two flashes. So instead of sampling the time uniformly, we sample it at discrete (separate and distinct) points. We can go on this way to create as many flashes as we desire:

A moving sphere rendered with discrete time sampling to simulate flashing lights

Whenever *RenderMan* needs a batch of camera rays, it goes through the built-in projection and calls the *Project* method of the active *RixProjection*. The plug-in receives the camera rays from the built-in one, nicely packed up in a *RixProjectionContext*. It can then iterate over these and make the desired adjustments. As an example, tinting all rays red and inverting their direction could be implemented in the *Project* method as follows:

```
// pCtx.numSamples contains the number of camera rays this
 ↪  RixProjectionContext contains
for (int i=0; i<pCtx.numSamples; ++i)
{
  // Tint rays red
  pCtx.tint[i] = RtColorRGB(1.0f, 0.0f, 0.0f);

  // Reverse direction or rays
  pCtx.rays[i].direction  = -pCtx.rays[i].direction;
}
```

For the multiflash effect, we're purely interested in adjusting the time of each ray. There are two fields concerning this aspect: *time* and *shutter*. Despite its name, the actual time associated with each ray is not represented by the *time* field but rather by *shutter*. *time* contains the raw samples, uniformly distributed between 0 and 1. *shutter* on the other hand, contains the remapped values to represent the actual time of the ray, respecting features like *shutteropening*. *time* is there only for your convenience, giving you access to the original random numbers. This is also why this field is defined *const* (ie. read-only), making *shutter* the only field we can control concerning a ray's time. For example, setting the *shutter* field to 0 effectively disables motion blur:

```
for (int i=0; i<pCtx.numSamples; ++i)
{
  // no motion blur, all rays sample the same time
  pCtx.shutter[i] = 0.0f;
}
```

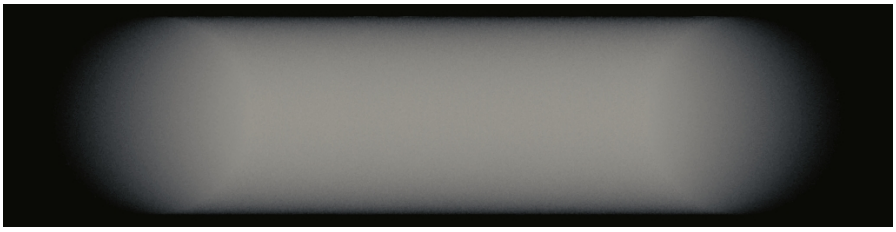
2.3 Implementation of the effect

With this knowledge, we have everything we need to implement the *Multiflash Lens*. To see the effects of our changes to the time-sampling, we need a *RIB* file with a moving object and motion blur. For this, I've updated our simple *RIB* file from above to include movement:

```
# Enable motion blur
Shutter 0 1

Hider "raytrace" "int minsamples" 32 "int maxsamples" 2048
Integrator "PxrPathTracer" "handle"

Format 1024 256 1
Display "preview" "it" "rgba"

# This is the builtin projection
Projection "orthographic"

# This loads our custom projection plug-in!
Projection "./MultiflashProjection"

ScreenWindow -4.5 4.5 -1.125 1.125

Translate 0 0 4
WorldBegin
  # Movement from x=-3 to x=3
  MotionBegin [0 1]
    Translate -3 0 0
    Translate  3 0 0
  MotionEnd

  Bxdf "PxrConstant" "constant"
  Sphere 1 -1 1 360
WorldEnd
```

Listing 3: SimpleRIB_motion.rib

You can copy-paste the *SimpleProjection* plug-in from above, rename any occurrence of *SimpleProjection* to *MultiflashLens* and store it in a new file called 'MultiflashLens.cpp'. Compile it, still with an empty *Project* method, and render the animated *RIB* from above. You should get a render of a motion blurred sphere:



A moving sphere rendered with our (still empty) MultiflashLens plug-in

Now, finally, we can go ahead and make changes to the *Project* method. We should be able to create the effect of two flashes by simply rounding the *shutter* value. Half of the rays will be rounded to 0 while the other half gets set to 1. Try adding the following to the body of *Project*:

```
int i=0; i<pCtx.numSamples; ++i)
{
  // std::floor(x+0.5f) rounds to its nearest integer value
  // This sets shutter '0' for half of the rays and to '1' for the others
  pCtx.shutter[i] = std::floor(pCtx.shutter[i]+0.5f);
}
```

However, when rendering with these changes, we only get the sphere rendered at time 0 instead of the two flashes we'd expect:

When rounding the shutter value to 0 and 1, we only get an image at time 0

The values for *shutter* have to be in the $[0, 1)$ interval, that is, any number between 0 and 1, including 0 and excluding 1. Setting *shutter* to 0 is legal while a value of 1 is not and makes the objects in motion disappear for these rays. So always make sure that your *shutter* values stay in the valid range:

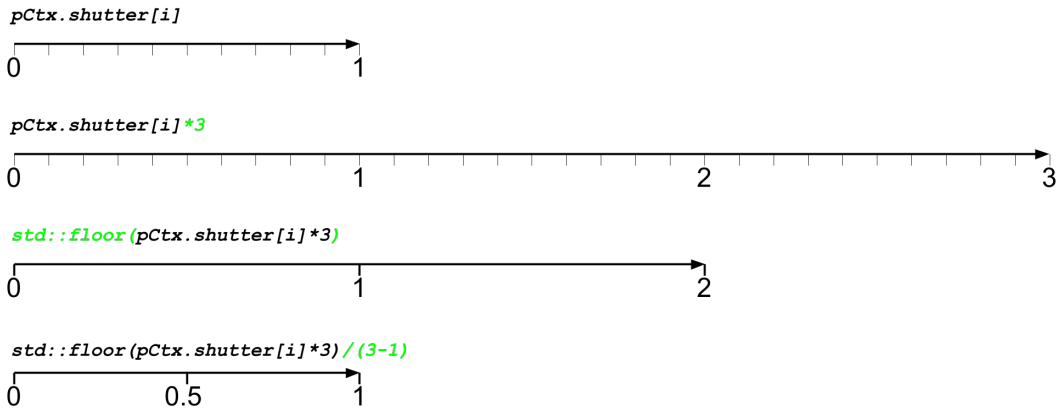
```
for (int i=0; i<pCtx.numSamples; ++i)
{
  // std::floor(x+0.5f) rounds to its nearest integer value
```

```
  // This sets shutter '0' for half of the rays and to '1' for the others
  pCtx.shutter[i] = std::floor(pCtx.shutter[i]+0.5f);

  // shutter has to be 0<=shutter<1! This statement clamps to make sure
  ↪  we're in the legal interval of [0,1)
  pCtx.shutter[i] = std::max(0.0f, std::min(1.0f-1e-7f,
  ↪  pCtx.shutter[i]));
}
```

Rendering this now gives us the two flashes we're looking for:

By keeping the shutter values in the valid range we now get the expected result

Adding more flashes is as easy as coming up with an algorithm which rounds the *shutter* values to multiple points between 0 and 1. If we "stretch" the *shutter* values before rounding and "squash" them back into the 0 to 1 range, we can get discrete points, evenly distributed in the desired range:

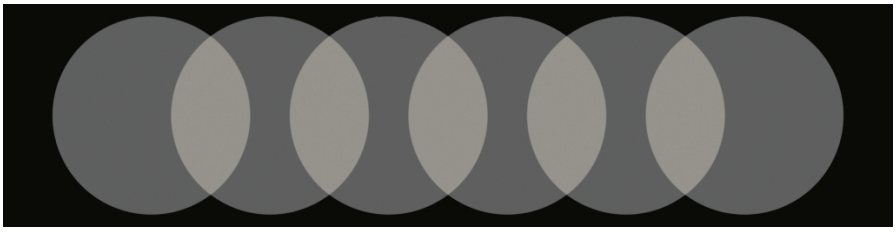


Algorithm to add an arbitrary amount of flashes. This example illustrates adding three distinct flashes

Translating this to code gives:

```
int numFlashes = 3;
for (int i=0; i<pCtx.numRays; ++i)
{
  pCtx.shutter[i] =
 ↪  std::floor(pCtx.shutter[i]*numFlashes)/(numFlashes-1);
  pCtx.shutter[i] = std::max(0.0f, std::min(1.0f-1e-7f,
 ↪  pCtx.shutter[i]));
}
```

With this, we can create any number of flashes we want, simply by changing the value assigned to the *numFlashes* variable. Here's a render using six flashes:



Using the described algorithm to render a moving sphere with 6 flashes

Excellent! You've successfully implemented a custom plug-in which simulates the multiflash effect!

2.4 Taking this into RenderMan for Maya

Rendering directly from a *RIB* file is very useful when developing. However, there is a point where we need to be able to test the plug-in in a production environment and more complex scenes. This is a good time to start applying our custom projection inside *RenderMan for Maya*.

The easiest way is to manually inject our custom projection call into the *RIB* generation. For this, we go to the *Advanced* tab in the RenderMan Render Settings dialog and add the following to the *Pre WorldBegin MEL*:

```
RiProjection("/path/to/MultiflashLens", "float dummy", 1);
```

When you're on *Windows*, don't forget to *escape* the backslashes (ie. every '\' becomes '\\'). The *dummy* parameter is necessary because the *RiProjection MEL* function expects at least one parameter, our plug-in is simply going to ignore it for now.

This approach is a very easy and unobtrusive (it doesn't affect the rest of your scene setup or configuration), making it my preferred choice when trying plug-ins in *RenderMan for Maya* during development or when I know it's only going to be used in a handful of scenes. There are other ways which would tightly integrate our custom plug-in and make it practically indistinguishable from the plug-ins *RenderMan* ships with but we'll save this for later.

With this, you can now take any of your animated scenes and apply the multiflash lens on it. Keep in mind that you might need quite high *Shutter Angle* values to get long enough trails. Also, make sure you increase the motion samples on objects with complex movements. This will increase your memory usage so keep an eye on that.

Here's a render I did of a simple billiard scene. This also includes some additions discussed in the next section:

Without the multiflash lens, this would just be a render with some blurry streaks. With our custom plug-in however, we see both movement and details, creating a unique and interesting look.

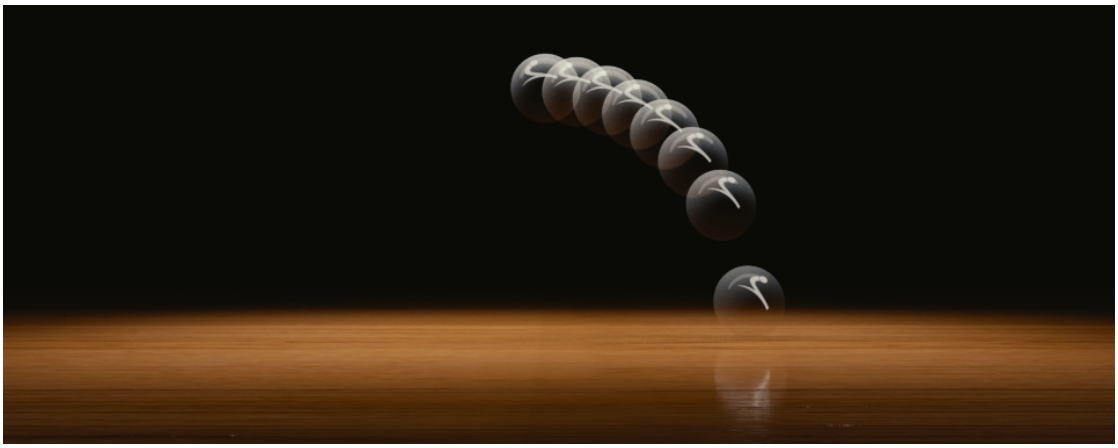
3 Going further

Now that we have implemented the basic multiflash functionality we wanted, it's time to get creative. *RenderMan* gives us a lot of control over a wide range of aspects in the

rendering process. But even with just *tint* and *shutter*, we can create a variety of effects.

3.1 Fading trail

One effect I particularly like is to make the first flashes less intense than the last ones, essentially making that 'trail' fade over time. This has also been used on the billiard render from above. Without it, we don't have any visual clues of the direction of motion:

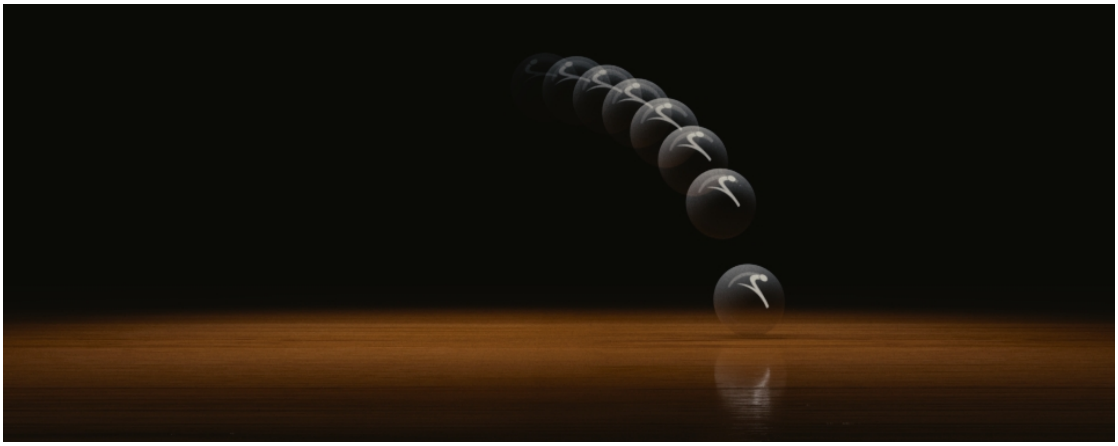


It's unclear whether the ball bounces from left to right or the other way around. Making the flashes at the beginning of the shutter darker than the ones towards the end will accentuate the direction of motion and make a more interesting picture.

What we need is a formula which assigns a dark tint to rays with a shutter value close to 0 and a bright tint to the ones close to 1. An easy way to achieve this is to simply convert the shutter value to a color. This gives us a dark tint for small *shutter* values and bright colors when we get closer to 1. Let's change our *Project* method to the following:

```
int numFlashes = 8;
for (int i=0; i<pCtx.numRays; ++i)
{
  // Converting shutter to a color, making rays with a low shutter dark
 ↪  and rays with a high shutter white
  pCtx.tint[i] = RtColorRGB(pCtx.shutter[i]);
  pCtx.shutter[i] =
 ↪  std::floor(pCtx.shutter[i]*numFlashes)/(numFlashes-1);
  pCtx.shutter[i] = std::max(0.0f, std::min(1.0f-1e-7f,
 ↪  pCtx.shutter[i]));
}
```

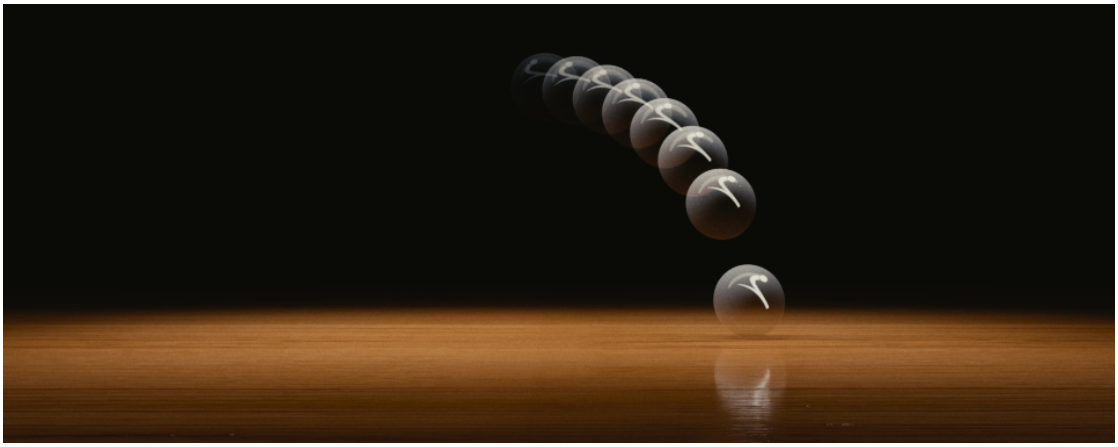
This gives the following render:



Do you notice that our render just got darker? This is because we're taking energy away from some rays without adding it back to others. On average, we're tinting our rays 50% grey. If we want to keep the same overall brightness, we need to average to white, so let's correct that:

```
int numFlashes = 8;
for (int i=0; i<pCtx.numRays; ++i)
{
  // We're multiplying with 2.0f to make sure we're maintaining the
 ↪  overall brightness
  pCtx.tint[i] = RtColorRGB(2.0f*pCtx.shutter[i]);
  pCtx.shutter[i] =
 ↪  std::floor(pCtx.shutter[i]*numFlashes)/(numFlashes-1);
  pCtx.shutter[i] = std::max(0.0f, std::min(1.0f-1e-7f,
 ↪  pCtx.shutter[i]));
}
```

Now we're back to the brightness levels we'd expect:



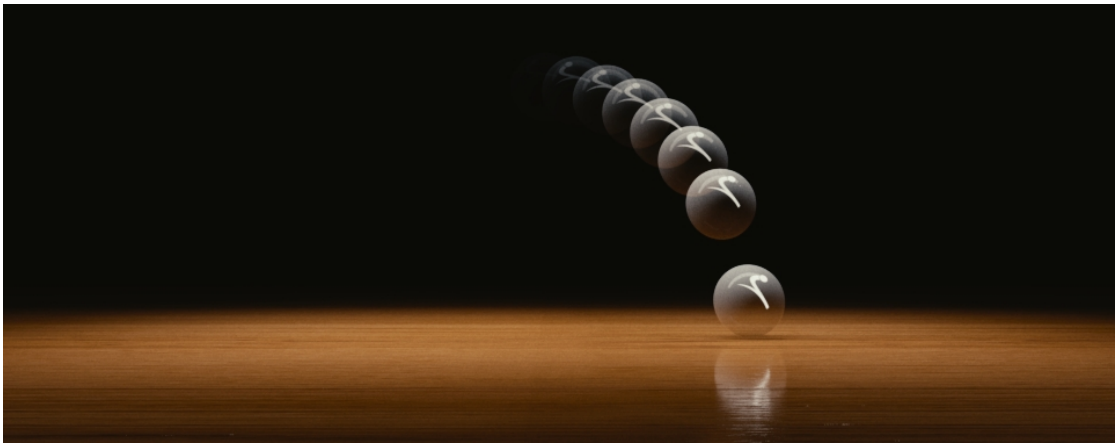
That's nice! See how we now get a feeling of motion from left to right? But what if we want to accentuate the motion even further?

We can use any formula we want, so it's really up to you. In my case, I just squared the *shutter* value and converted it to a color. This makes the flashes start dim and then brighten up rapidly towards the end of exposure. If we again want to maintain the overall brightness we have to make the function averages to white. I'll do the maths here in case you're interested.

Our function essentially is x^2 and our goal is to make it average to 1 over the shutter interval $[0, 1)$. So, we're looking for a value c for which $\int_0^1 cx^2 dx = 1$. Solving the integral this gives: $\frac{1}{3}c = 1$ and thus $\underline{\underline{c = 3}}$.

Applying this new formula in the code gives:

```
int numFlashes = 8;
for (int i=0; i<pCtx.numRays; ++i)
{
  pCtx.tint[i] = RtColorRGB(3.0f*pCtx.shutter[i]*pCtx.shutter[i]);
  pCtx.shutter[i] =
↪   std::floor(pCtx.shutter[i]*numFlashes)/(numFlashes-1);
  pCtx.shutter[i] = std::max(0.0f, std::min(1.0f-1e-7f,
↪   pCtx.shutter[i]));
}
```



Great! This shows a clear direction of movement simply by adjusting the brightness of our flashes, all this by just adding a single line of code to our plug-in!

4 Conclusion and final remarks

With just a small amount of programming, we've been able to use *RenderMan*'s *Rix-Projection* interface to create a unique look which would have been difficult to achieve out of the box. With small additions and adjustments in the code we're able to create a wide range of looks and take control over the result of our render. All this with just a handful of lines of code, imagine what else you could do!

5 Acknowledgment

I thanks Dylan Sisson and Chris Ford for making it possible for me to write these lessons and for their support.

Further, I thank Chu Tang, Leif Peterson and Greg Shirah for proof-reading and their great suggestions.