Technical Report TR-2014-06

CHRONO::RENDER A GRAPHICAL VISUALIZATION PIPELINE FOR MULTIBODY DYNAMICS SIMULATIONS

Daniel Kaczmarek, Aaron Bartholomew, Felipe Gutierrez, Hammad
Mazhar, Dan Negrut

May 22, 2014

Abstract

This paper describes a web-enabled tool capable of generating high quality videos and images from multibody dynamics simulation results. This tool, called Chrono::Render, uses the Blender modeling software as the front end with Pixars RenderMan used to create high quality images. Blender is a free and open source tool used to create and visualize 3D content and provides a robust plugin framework which Chrono::Render leverages. To produce the final image, the Blender front end passes data to a RenderMan compliant rendering engine. Along with Pixars PhotoRealistic RenderMan (PRMan), several open source options such as Aqsis, JrMan, or Pixie can be used. Pre-processing is performed on the client side, where the front end generates a work order for the RenderMan compliant rendering engine to process. This work order, which contains several scripts that define the visualization parameters, along with the pre-processed simulation data and other user-defined geometry assets is uploaded to a remote server hosted by the Simulation Based Engineering Lab. This server contains more than a thousand CPU cores used for high performance computing applications, which can be used to render many frames of an animation in parallel. Chrono::Render is free and open source software released under a BSD3 license.

Keywords: Chrono::Render, rendering, RenderMan, Blender

Contents

1	Introduction	3
2	RenderMan	5
3	Blender Plugin	6
4	Chono::Render Post-Processor	8
5	Server Side Chrono::Render	9
6	Results	9
6.1	Waterwheel	9
6.2	Granular Material Dragon	11

Figure 1: Simulation of foam rendered with Chrono::Render.

1 Introduction

Chrono::Render [18] is a software package that enables fast, simple, and streamlined visualization of simulation data using Pixar’s PhotoRealistic RenderMan (PRMan) [13]. The idea of Chrono::Render is to abstract away the complexities of rendering a complex scene. To this end, Chrono::Render provides a Graphical User Interface (GUI) using Blender [6], an open source 3D modeling program which generates configuration files that Chrono::Render processes and sends to RenderMan to generate the final image, Fig 1.

The process of rendering involves generating an image based on a description of a scene. This description usually contains information about objects, lights, cameras, and the environment. Rendering is a time consuming process and can sometimes take longer than the simulation as the process of generating realistic images involves the simulation of millions of light rays or photons to accurately capture how light behaves. Despite its complexity, rendering can be easily parallelized as individual images are not dependent on one another. If enough compute power is available, many images can be rendered at once, greatly decreasing the time needed to generate an animation.

There are many commercial and open source tools available for visualization purposes.

However, many of these tools are not capable of efficiently visualizing simulation data, as the type of data is highly dependent on the type of simulation. There is no single data format that can encapsulate all of the required information, meaning that the software must provide a way to process custom information. Tools such as Lightwave [11], Autodesk 3ds Max [2], and Autodesk Maya [3] have their own scripting interfaces [4, 5, 12] or expose Python [16] modules that can interact with the software. Through these scripting interfaces custom data can be parsed, imported, and used to generate and place geometry in a scene. One major drawback to this approach is that large sets of simulation data with millions of objects usually require too much memory and cannot be imported in such a manner. The default rendering engines provided by these tools are not designed to render large complex scenes and will often either fail to render a scene in a reasonable amount of time or will exhaust the available memory during the rendering process. Rendering engines such as Persistence of Vision Raytracer (POV-Ray) [19] are very suitable to rendering large sets of simulations data. POV-Ray uses a ray tracing algorithm to render images; this approach is powerful because it uses analytical representations of geometry during the rendering process, e.g., a sphere is defined by its position and radius. In comparison, tools such as Autodesk Maya use a polygonal representation where each sphere is decomposed into hundreds of small triangles, with a higher triangle count leading to a smoother sphere, but also more memory usage. The analytical representation allows POV-Ray to render extremely large sets of simulation data with a small memory footprint, however, image quality is often lacking as newer and more sophisticated algorithms for rendering exist. For this reason RenderMan was selected as it is independent of any modeling software and is the state of the art in terms of render quality and the amount of control that is provided to the user. Similarly to POV-Ray, it also uses analytical representations of geometry but has the added bonus of utilizing the latest improvements in the computer graphics field. Furthermore, RenderMan is capable of rendering large sets of simulation data with a small memory footprint, making it an ideal choice for the rendering engine.

Chrono::Render provides rendering as an automated post-processing step in a remote simulation pipeline. It is controlled via a succinct specification provided using the YAML [8] syntax language as well as optional user-defined Python scripts which are combined to allow a visually-rich scene to be rendered by RenderMan. This pipeline offers tools for the visualization of simulation data through the use of a RenderMan compliant renderer [14]. Typically, in order to use a RenderMan compliant rendering engine, a significant amount of knowledge in the fields of computer graphics and programming is required. This barrier to entry makes it difficult for many people to generate high quality visualizations from their simulation data. Chrono::Render is designed to reduce this barrier and eliminate the need for sophisticated knowledge of computer graphics while allowing users to leverage the powerful rendering engines. There are three main components to Chrono::Render: the Blender GUI, post-processing scripts, and a web interface that allows jobs to be submitted remotely. These three components are independent with only the post-processing scripts needed to render an image, Fig 2.

Figure 2: Rendering pipeline workflow.

2 RenderMan

RenderMan is an application programming interface (API) provided by Pixar for visualizing complex scenes that involve millions of polygons, many lights, and surfaces that are transparent, reflective, or ones that refract light. The specification that defines a RenderMan compliant renderer is defined in the RenderMan Interface Specification (RISpec) [14] which is an open specification that several rendering engines have implemented. RenderMan is designed to be scalable across hundreds of compute nodes and thousands of processor cores. Individual frames in an animation are divided into small sections that are processed in parallel. By leveraging newer multi and many-core CPU architectures, high resolution images can be generated in hours instead of days. There are several open source RenderMan compliant rendering engines including Aqsis [9], jrMan [10], and Pixie [1]. In terms of closed source options, Pixar’s PhotoRealistic RenderMan (PRMan) [13] is the most powerful option available and is widely used throughout the computer graphics industry. Due to its maturity and the features it provides, PRMan is the default rendering engine used by Chrono::Render for rendering images. The aforementioned rendering engines all use the REYES (Renders Everything You Ever Saw) [15] architecture that involves dividing every surface that will

be visualized into micro-polygon grids. These small sets of geometry can then be processed concurrently across multiple cores of a CPU with a small memory footprint, greatly improving the speed at which an extremely large scene with millions of objects and hundreds of millions of triangles can be rendered.

In order for a RenderMan compliant renderer to generate an image it requires two types of information. First is the RenderMan Interface Bytestream (RIB), this is the file format that RenderMan uses to generate and visualize a scene. Information about the locations and rotations of the camera, lights, and simulation objects are all contained in these files. The bytestream can be stored in a file and input into RenderMan or it can be generated dynamically, stored in memory, and passed to RenderMan. The latter of the two options is more efficient and allows a RIB file to be generated on the fly reducing the amount of disk space and time needed to render. The second piece of information that RenderMan requires is the material properties of an object. These properties are defined in a *shader* file, see Appendix E, which defines how light interacts with that material. Properties such as object color, reflectance, transparency, etc. are used to compute the final color of an object.

Generating the RIB and shader information is not trivial and requires extensive knowledge about computer graphics. In order to simplify this process Chrono::Render provides a Blender GUI which handles the generation of this information.

3 Blender Plugin

Chrono::Render leverages the visualization capabilities already available in Blender and adds pre-processing support for simulation data through a custom plugin. This plugin works with Blender using its python API [7] to provide a low-quality real-time view of the simulation data as well as a robust and well-documented interface for finalizing a scene. Assets such as materials and lights, which are generally not provided in the simulation data, are added using this interface, Fig 3.

This plugin works by importing and exporting data files containing data for visualizing the simulation using a custom file format, the specification for which can be found in Appendices A and C. The plugin supports a number of commonly used primitives including spheres, cubes, ellipsoids, cylinders, cones, and arbitrarily dimensioned boxes. In addition to these primitives, the plugin provides support for importing arbitrary meshes using the wavefront obj file format.

An important feature of this plugin is the robust handling of millions of primitive objects for scenes dealing with large-scale granular dynamics problems or fluid flow. Real-time visualization of such scenes is not practical in Blender due to the associated cost in memory. This issue is solved by using *proxy* objects where sets of similar geometry that share material or size information are grouped together and displayed using a representative geometry. The proxy group used for a set of proxy objects is specified in the file format in which the first parameter on each line provides the proxy group identifier that the plugin uses to generate proxy objects for the scene. It is also possible to make an object not part of a proxy group by

Figure 3: Exporting a scene with the Blender plugin.

setting the first parameter to "individual", in which case the object will be directly visible in Blender.

Simulations with millions of objects can be simplified into several proxy objects that the user can apply material and other visualization properties to. Individual objects such as static walls and custom meshes are visualized separately. The combination of individual and grouped objects aims to provide enough information to position the camera and lights, but prevent an overload of information that would make visualization unwieldy and slow.

Apart from providing a real time interface for visualizing and setting up the simulation, this plugin is also used to prepare the scene for final rendering using a RenderMan compliant renderer. Exported data includes information about the camera (see Appendix D), lights (see Appendix C), object visibility, shadow data, image resolution, object color, camera key-framing, ambient occlusion, and color bleeding. This data is compressed into two archives. The first contains the simulation data as specified in Appendix A. The second contains Blender generated configuration files specifying how the scene will be rendered. These are human readable so in the event that the user requires modifications not supported by the Blender interface, they can still make them manually in the configuration files. This could occur in cases where features not yet implemented in the plugin are desired (e.g. applying a shader), or for parameters that RenderMan possesses but Blender lacks. Because the simulation data is stored in a separate archive, changes to the configuration files do not require the data to be re-uploaded. Often the simulation data can reach hundreds of megabytes or gigabytes while the configuration files are typically only on the order of kilobytes. The two

archive format allows for quick changes to the lighting and camera setup, without requiring the re-upload of potentially gigabytes of data each time.

Setting the camera and lights within Blender allows the user to take advantage of Blender’s internal rendering engine for immediate visualization. Rendering a scene is often time consuming and during the setup of the visualization it is useful to have immediate feedback on whether the lighting looks correct or if the camera is positioned properly. Blender’s built-in rendering engine provides this immediate visualization, giving a rough render of the scene within Blender itself. While Blender’s built-in engine is not capable of visualizing the whole scene, it is capable of visualizing the key elements of the scene, and provides useful feedback on lighting and camera positioning.

4 Chrono::Render Post-Processor

The final stage of the visualization process involves the Chrono::Render post-processor. This software parses the data and configuration files provided by the Blender plugin, and generates the RIB files for RenderMan. Each RIB file is independent of the rest and defines the information required by a RenderMan compliant renderer to render one frame of animation.

The main goal of Chrono::Render is to reduce the barrier to entry and provide a visualization pipeline that does not need sophisticated knowledge of computer graphics. However, in certain cases the user may want more control over the RIB file generation and rendering processes. Chrono::Render supports this by allowing the user to write and execute their own python scripts that are capable of manipulating the RIB files directly. Using this feature requires knowledge of the rendering process, but it allows more advanced effects such as varying the object color or shape based on various properties as well as the ability to visualize information such as stress, velocity and pressure fields.

While the Blender plugin automatically generates the configuration files for Chrono::Render, manual editing provides the user with a finer degree of control. This allows the user to modify values that are not supported by the Blender plugin such as custom materials and a number of other parameters that control the render time/quality trade-off. Properly tuning these values for a simulation can reduce render times by several hours.

The main configuration file generated by Chrono::Render is output in the human readable yaml [8] format. The automatically generated settings are typically acceptable, however, there are several useful parameters that can be modified. These parameters come into play when generating a render pass. An image is generated using several passes, which are combined to form the final image. For example, in order to add shadows to an image, a shadowpass must be specified. The quality of this pass can be specified by providing a resolution, where higher resolutions result in better shadows at the cost of increased render time. Another parameter controls the shading rate of the renderer, with a smaller value resulting in a higher quality image, while a larger value meaning the image will be of lower quality but generated faster. Increasing the pixel samples parameter will increase quality, but the renderer will take considerably longer to finish an image. More information on these parameters can be found in the RenderMan specification [14].

Material properties are controlled using shaders - small pieces of code that specify how light interacts with an object and provide information such as color, specularity, and reflectivity to the rendering engine. Custom shaders can be specified for each object or group of objects. A default matte shader is provided with Chrono::Render, along with a library of other basic shaders including glass, plastic, and metal. If needed, additional shaders can be included in the archive containing the configuration files.

5 Server Side Chrono::Render

Chrono::Render offers a client-server model to allow remote renderings of simulations. The server side Chrono::Render consists of the post-processing scripts which generate the RIB files from the archives generated by Blender, as well as the RenderMan compatible renderer. The client interacts with the server side Chrono::Render via a python web interface which allows a user to upload the two archives containing the scene data and metadata, as well as set rendering parameters such as job name, walltime, number of cores, etc. Note that the user is responsible for hosting these two archives as the web interface automatically downloads them to the server. To this end, the user must provide a web address to where the archives are stored. Once the upload is complete the two archives are processed by the server side Chrono::Render software which starts the rendering process. The status of the job is constantly monitored using the PyTorque resource manager [17]. An email is sent to the user with a link to the final images/video once rendering is completed.

6 Results

In this section results from different simulations will be provided to demonstrate the types of images that Chrono::Render can generate. The results were rendered using the Simulation Based Engineering Laboratory's cluster.

6.1 Waterwheel

This image, Fig 4, shows a simulation of a waterwheel with over 64,000 individual spheres. The spheres and paddles of the waterwheel each have their own proxy objects while the cylindrical parts of the waterwheel along with the boxes are represented individually in Blender for easier positioning of camera and lights, as well as to enable their individual coloring. The rendering takes under two and a half minutes while using eight cores. The image uses no real shadows, instead making use of ambient occlusion to create a shadow-like effect, and uses a standard matte shader for all components of the simulation.

Figure 4: Render of a waterwheel simulation containing over 60,000 spheres

Figure 5: Render of a simulation containing over 180,000 spheres.

6.2 Granular Material Dragon

This image, Fig 5, shows a dragon comprised of 180,000 individual spheres. Because the spheres share a common material, they are imported into Blender as one proxy object, with the floor of the simulation remaining separate for camera placement and coloring purposes. The rendering takes under four minutes while using eight cores. The image forgoes fancy techniques, using only simple shadowmaps and a basic matte shader. Velocity data for each sphere can also be visualized with some python scripting, which provides a useful way to visually debug simulations.

Conclusion

This document describes a rendering tool capable of generating high quality videos and images from the results of multibody dynamics simulations. This tool leverages RenderMan to visualize complex scenes with millions of objects. Chrono::Render provides a user friendly interface via a Blender plugin that a user can use to convert simulation data into a RenderMan compliant format. Using a post-processor the process of data generation and rendering are separated, enabling Chrono::Render to provide a server side rendering service with a web interface that allows users to generate data locally and upload it to a server to render on a many core cluster. This flexibility means that Chrono::Render can be incorporated into many different work flows and does not require the user to have a powerful laptop or desktop for visualization purposes.

Work on Chrono::Render is ongoing and additional features to the Blender plugin will allow for dynamic light and camera movement. Additionally many of the advanced features that are not provided by the plugin could be exposed to the user in the future. The web interface to the server side Chrono::Render is functional yet limited. Improving this interface and providing more information to the user about how a render job is being completed would make it a more powerful tool.

References

- [1] Okan Arikan. Pixie. <http://www.renderpixie.com/>, 2014.
- [2] Autodesk Media and Entertainment. Autodesk 3ds max. <http://www.autodesk.com/3dsmax>, 2014.
- [3] Autodesk Media and Entertainment. Autodesk maya. <http://www.autodesk.com/maya>, 2014.
- [4] Autodesk Media and Entertainment. Max script. <http://docs.autodesk.com/3DSMAX/16/ENU/MAXScript-Help/>, 2014.

- [5] Autodesk Media and Entertainment. Maya embedded language - mel. http://download.autodesk.com/global/docs/maya2014/en_us/Commands/index.html, 2014.
- [6] Blender Foundation. Blender. <http://www.blender.org/>, 2014.
- [7] Blender Foundation. Blender python api. http://www.blender.org/documentation/blender_python_api_2_69_10/, 2014.
- [8] Clark C. Evans. Yaml: Yaml ain't markup language. <http://www.yaml.org/>, 2014.
- [9] Paul Gregory. Aqsis. <http://www.aqsis.org/>, 2014.
- [10] Gerardo Horvilleur, Jorge Vargas, Elmer Garduno, and Alessandro Falappa. jrman. <http://www.jrman.org/>, 2014.
- [11] NewTek, Inc. Lightwave 3d. <http://www.lightwave3d.com/>, 2014.
- [12] NewTek, Inc. Lightwave scripting language. <ftp://ftp.newtek.com/products/LightWave/SDK/LScriptUserGuide.pdf>, 2014.
- [13] Pixar. Photorealistic renderman (prman). <http://renderman.pixar.com/view/renderman>, 2014.
- [14] Pixar. Renderman interface specification. <http://renderman.pixar.com/view/rispec>, 2014.
- [15] Pixar. Reyes - renderman under the hood. <http://renderman.pixar.com/view/reyes-renderman-under-the-hood>, 2014.
- [16] Python Software Foundation. Python. <http://www.python.org/>, 2014.
- [17] Alexander Rauch. Pytorque. <https://github.com/rauch/PyPBS>, 2014.
- [18] SBEL. Chrono::render. <http://projects.sbel.org/ChronoRender/>, 2014.
- [19] The POV-Team. Persistence of vision raytracer - pov-ray. <http://www.povray.org/>, 2014.

Appendix A: File Format

Each file represents a frame, each line represents an object. The format of the lines are:

Group Object ID, xpos, ypos, zpos, quatw, quatx, quaty, quatz, object type, extra params

Group represents the group name and identifies what objects form each proxy object. The Object ID is a unique identifier of the object; no two objects have the same ID. The parameters xpos, ypos, zpos are the coordinates of the object and parameters quatw, quatx,

quat, quatx are the rotation of the object in quaternions. Object type represents what kind of object it is, current options include: sphere, cube, box, cylinder, and cone. Extra parameters are specific to the object type. For example, a sphere has only one extra parameter: the radius, while a cone requires both the radius and the height. All extra parameters are comma delimited like the rest of the file.

Appendix B: Lighting

The plugin supports the Blender point, sun, spot, and ambient lighting types. The relevant parameters from these lights are converted into the RenderMan lighting types: point light, distant light, spotlight, and ambient light respectively.

With the exception of ambient lighting, all other lights use the default Blender controls. This is because Blender uses a per-object parameter to measure ambient intensity whereas RenderMan uses a single parameter for all objects. Therefore, the export to RenderMan will use the ambient attribute of the Ambient Light Proxy Object for the intensity of all objects instead of using their individual parameters. The listing below provides an example of how lighting information is specified in a RIB file. There are two lights, the first casts a shadow and the second provides ambient light.

```
LightSource
"shadowdistant" 7
"intensity" 0.75
"lightcolor" [1.0 1.0 1.0]
"from" [0 0 0]
"to" [0.0 -1.0 0.0]
"shadowname" ["shadow_Lamp.shd"]
```

```
LightSource
"ambientlight" 8
"intensity" 1.0
"lightcolor" [0.0 0.0 0.0]
```

Shadows are done by going through a separate render pass for each light in order to create shadow maps. A shadow map is a texture that stores the shadows generated by a single light. During the rendering process, shadow maps are used to darken the areas where there are shadows. Shadow maps produce photo-realistic shadows and are commonly used in the computer graphics industry. The code listing below provides example RIB code for a shadow light.

```
LightSource
"shadowspot" 9
"intensity" 74.7000002861023
"coneangle" 0.5846853256225586
"conedeltaangle" 0.5846853256225586
```

```

"lightcolor" [1.0 1.0 1.0]
"from" [0.0 2.9700000286102295 7.699 999809265137]
"to" [0.0 2.3486650586128235 6.91645473241806]
"shadowname" ["shadow_Spot.shd"]

```

An alternate method of generating shadows is to use ambient occlusion. This method uses ray tracing to approximate a shadow effect. While outside the scope of this document, the process involves rays that are shot out from a point, with the darkness of the object determined by whether or not the ray hits another object or not. This effect darkens the areas near multiple surfaces as often witnessed in real life. While the effect is not photo-realistic, it often times provides good approximations and can be used both by itself and in combination with shadowmaps.

Appendix C: YAML configuration file

The listing below provides a full example of a YAML configuration file for Chrono::Render. In this file the locations of the camera and lighting information are specified while the render pass information is specified directly in the file. There are two render passes, one for shadows and a second pass to compute ambient occlusion information. Information about how the data is organized is provided along with the materials and shapes used in the scene. Notice how the shader from Appendix E is specified in the last line of the file.

```

chronorender:
  camera:
    - {filename: custom_camera.rib}
    - {moving_camera: false}
  lighting:
    - {filename: custom_lighting.rib}
  renderpass:
    - name: shadowpass7
      settings:
        display: {...}
        pixelsamples: 1 1
        resolution: 512 512 1
        shadingrate: 1.0
        shadowfilepath: shadow_Lamp.rib
      type: shadow
    - name: ambientpass
      settings:
        bounces: 3
        display: {output: out.tif}
        resolution: 1920 1080
      shader: {...}

```

```

    type: ao
rendersettings: {searchpaths: ./}
simulation:
  data:
    datasource:
      - fields:
          - [group, string]
          - [id, integer]
          - [pos_x, float]
          - [pos_y, float]
          - [pos_z, float]
        name: defaultdata
        resource: ./data/data_*.dat
        type: csv
  renderobject:
    - color: 1.0 1.0 1.0
      condition: id == 4
      geometry:
        - {changingprams: ...}
      name: individual
      shader:
        - {name: matte.sl}
    - color: 1.0 1.0 1.0
      condition: (...)
      geometry:
        - {changingprams: ...}
      name: g0
      shader:
        - {name: matte.sl}

```

Appendix D: Camera RIB

A typical RIB file to specify the camera is provided in the listing below. The "fov" parameter provides the field of view, or how wide of an angle the camera sees. Transformations such as scaling, rotating and translating can be provided and will be applied to the camera in the order specified. A negative scale can be used to mirror the camera along a certain axis. This file is typically generated by the Blender plugin. If a moving camera is desired, separate files will be generated for each frame of the simulation.

```

Projection "perspective" "fov" [39.122]
Scale 1 1 -1
Rotate -0.0 1 0 0

```



```
Rotate -0.0 0 1 0
Rotate -0.0 0 0 1
Translate -0.0 3.8 -12.0
```

Appendix E: Example Shader

Shader files encompass the information about a material applied to an object. The shader provided below is one of the default shaders that is provided with RenderMan and is capable of providing a "matte" finish to an object.

```
/* matte.sl - Standard matte surface for RenderMan Interface.
 * (c) Copyright 1988, Pixar.
 *
 * The RenderMan (R) Interface Procedures and RIB Protocol are:
 * Copyright 1988, 1989, Pixar. All rights reserved.
 * RenderMan (R) is a registered trademark of Pixar.
 */

surface matte (float Ka = 1;
               float Kd = 1;)
{
    point Nf;
    Nf = faceforward (normalize(N), I);
    Oi = Os;
    Ci = Os * Cs * (Ka * ambient() + Kd * diffuse(Nf));
}
```