SAND2019-12756C

An Optical Test Simulator Based on the Open-Source Blender Software

Daniel P. Rohe

Sandia National Laboratories*
P.O. Box 5800 - MS0557
Albuquerque, NM 87123
dprohe@sandia.gov

ABSTRACT

With camera equipment becoming cheaper and computer processing power increasing exponentially, optical test methods are bound to become ubiquitous in the structural dynamics community. However, unlike more traditional methods where the measurement response of interest is obtained directly from the sensor (e.g. an accelerometer directly provides an acceleration), image-based measurement techniques often require a non-trivial amount of post-processing to extract displacements from a series of images. Using experimental images to develop and validate these post-processing algorithms can be a challenge; real images have a finite depth of field, they can have poor contrast, they can be noisy, there can be calibration errors, etc. It would be advantageous to create an optical test simulator in which image processing algorithms could be investigated without the need to deal with all the complexity and cost involved in a real experiment. A simulator would also provide access to a "true" analytical solution, which is often not available in an experiment. It could also be useful for test planning purposes. Blender is a free and open-source 3D software package that supports modeling, rendering, and many more features that are not relevant to this work. It runs an underlying Python scripting engine, so simulator activities such as building or deforming a mesh can be automated. This work will demonstrate Blender's suitability for use as an optical test simulator and show an example workflow.

Keywords: Blender; Optical; Simulation; Analytical; Finite Element;

1 Introduction

Setting up an optical structural dynamics test is not trivial. A high-speed camera or two are required. They need to be attached to a rigid support so they stay stationary. They need to be calibrated. Lighting must be adequate for the shutter speed required for the test. The part needs to be prepared with some kind of contrast pattern. Excitation must be delivered to the part and synchronized with the cameras. And after all this effort, there may not be any truth data to compare the measured response against. Synthetic images offer a way to develop and validate experimental techniques without the complexity and potential error sources inherent in an experiment.

Initial efforts in generating realistic synthetic images grew from computer graphics applications. For example, starting with a simple pinhole camera model, Potmesil and Chakravarty described adding lens and aperture effects to simulate depth of field [1] in order to make more realistic renders. As optical testing techniques evolved, the need for simulated or synthetic images to understand the uncertainty in Digital Image Correlation (DIC) or other optical techniques was identified by a number of authors. Orteu, et al. investigated the generation of speckle textures to generate realistic images for DIC studies [2]. Garcia,

*Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

et al. proposed a generic synthetic image generator to further investigate uncertainty in DIC and computer vision techniques, considering cameras, textures, and lighting, and utilizing continuous functions rather than finite element meshes to define object surfaces [3]. Other authors have proposed similar simulators to investigate DIC that operate using finite element meshes, for example to generate synthetic images that can help understand the effects of large, heterogeneous displacement fields [4], identify material properties [5], or understand calibration uncertainty [6]. This work will demonstrate a similar capability of simulating an optical test; however, the capability will be built within a mature open-source framework with a graphical user interface requiring minimal additional coding, except where automation is desirable. With the optical simulator in place, it can be used to test experimental techniques on "perfect" images, understand uncertainty and errors involved with specific experimental issues, and plan for tests given test geometry, camera resolution, and lenses available.

2 Blender Overview

Blender[1] is a "free and open source 3D creation suite. It supports the entirety of the 3D pipeline—modeling, rigging, animation, simulation, rendering, compositing and motion tracking, even video editing and game creation. Advanced users employ Blender's [Application Programming Interface (API)] for Python scripting to customize the application and write specialized tools" [7]. Blender's primary use is in computer graphics applications. Blender has been used to create photo-realistic renders of scenes, with capabilities including complex lighting, depth of field, volumetric effects (fog/mist), and physically accurate reflections and refractions. However, it has also been used in scientific and engineering applications to produce synthetic images, primarily for computer vision and machine learning applications. For example, in [8], the authors create a dataset of images of humans using Blender that can be used to train computer vision systems to recognize human poses. In another example [9], a series of photo-realistic bathroom scenes were rendered to train the computer vision system in an autonomous cleaning robot to recognize a toilet seat.

In this work, only a subset of the features of Blender will be used. The Python programming interface built into the software will be used to automate the creation of a Blender mesh from finite element nodal coordinates and element connectivity matrices. Materials will then be applied to the mesh using Blender's graphical user interface, and cameras and lights will then be placed in the scene. Finally, the mesh will be deformed and images will be rendered of the scene; a Python script automates this step as a large number images may be desired.

Blender is a powerful piece of software with many features that are out of scope of this work. This work will instead focus on features of the Blender software which are useful for simulating a structural dynamics test. Key portions of the Blender data model are enumerated below:

- Scene – A Scene in Blender is a 3D space in which Objects can be positioned.
- Object – An Object in Blender is an abstract "container" that holds other data (meshes, cameras, lights, etc.). Key features of an Object are position and orientation in space, as well as the data contained within it.
- Mesh – A Mesh in Blender is a set of vertex locations and a connectivity array that describes how the vertices are connected into edges and faces.
- Camera – A Camera in Blender specifies a viewpoint of the scene and how it is rendered. Parameters such as lens focal length and aperture can be specified.
- Lamp – A Lamp in Blender acts as a light source for the scene. Lamps come in a variety of types, such as spotlight or point light. Light can also be provided by a Mesh with an emissive material, which is useful for simulating area lights such as windows or large lamps.
- Material – A Material in Blender is a model of how light reacts to a Mesh's surface when it interacts with it.

Blender stores all of these items and others in its database, and it links the items together. For example, to create a Mesh in a Blender Scene, the created Mesh is first linked to an Object, and then that Object is linked to the Scene. Materials may be linked to the mesh as well. Figure 1 shows the Blender Outliner window which an outline of the scene that will be created in

[1]All work in this report was performed with Blender version 2.79. Shortly after this work was performed, Blender version 2.80 was released which included a number of user interface and render engine changes. If using this paper as an introduction to Blender, it may be helpful to download and install version 2.79 rather than 2.80.

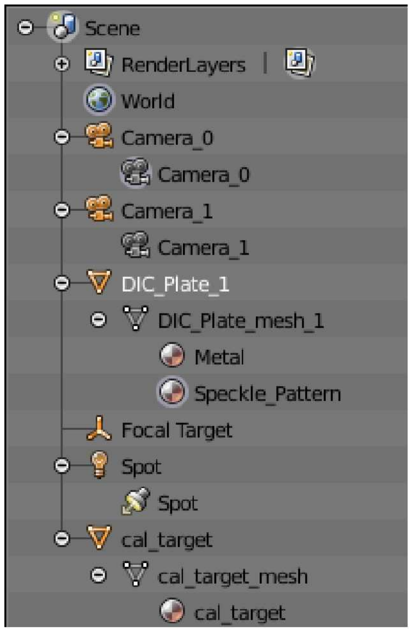


Figure 1: Outline of the scene that will be created in Section 3.

this paper, in which several objects are linked to the scene, and each object contains underlying data (Camera, Mesh, Lamp, etc.). Each mesh also has one or more materials linked to it. The scene also contains global World and Render information which will not be discussed in depth in this paper.

This underlying data model is hidden from the user interacting through the Graphical User Interface, but if utilizing the programming interface the user must understand the data model. Except for very specific cases, every scene should have at least one Mesh, light source, and Camera. Without a Camera, there will be no viewpoint from which to render the scene. Without a light source, everything will render black, and without a Mesh, there will be nothing in the scene to render.

A good portion of the work involved in setting up a Blender scene involves working with Materials and Lighting to achieve the desired result. Photorealistic images are not strictly necessary for the optical test simulator application, but there are very practical implications of the materials and lighting that will influence the results of a test. For example if the lights are too bright, the pixel intensities will clip. If there is not enough contrast in an applied texture, optical techniques will suffer. If we are dealing with a glossy object that may have specular reflections, we should simulate that as realistically as possible to understand what data may be lost.

Currently, Blender has two render engines, the Blender render engine (often called the “Internal” render engine, which this paper will use to avoid confusion between the software package and the render engine) and the Cycles render engine. The Internal engine was the first render engine packaged with Blender, and it generally produces less photo-realistic results but at reduced computational cost. The Cycles engine was added subsequently and is a more photo-realistic render engine at an increased computational cost. The Internal engine is a software renderer, but the Cycles renderer can be run directly on the Graphical Processing Unit (GPU) if the hardware is supported by Blender. This allows Cycles to be just as fast as the Internal renderer for many applications. While Objects and Meshes in the scene are generally compatible with both render engines, materials will generally need to be remade when swapping between render engines due to a fundamental difference in how the materials are used.

For more information regarding these or any other Blender topics, there is a wealth of information on the Internet. Of particular use are the Blender StackExchange site[2] and the Blender Documentation site[3], though many Blender forums, videos, and tutorials for using Blender exist throughout the Internet.

[2]https://blender.stackexchange.com/
[3]https://docs.blender.org/

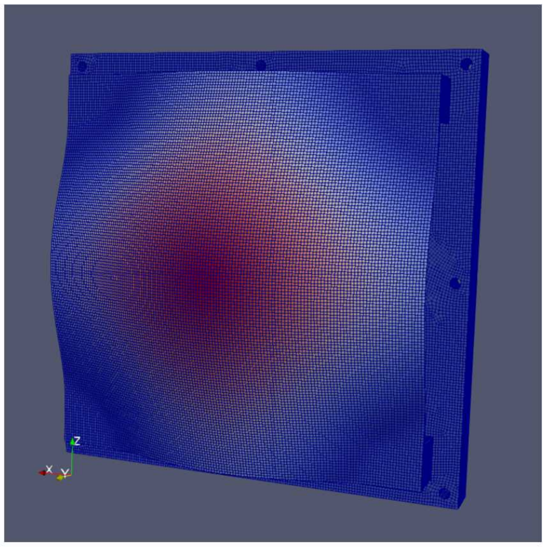


Figure 2: First bending mode of Plate structure

3 Generating Digital Image Correlation Images: Example Workflow

This section will describe a typical workflow using Blender to produce a set of test images for analysis in a DIC software package. A true tutorial showing every keystroke and mouse click is outside the scope of this work; accordingly, this example will focus instead on the general process.

3.1 Preliminary Finite Element Manipulations

A finite element model of a plate structure was created, and an eigensolution was performed to solve for the modes. Figure 2 shows the finite element model deformed in its first bending shape. To generate displacement time history data for DIC simulation, an analytical impact was applied to the corner of the plate, and the modal degrees of freedom were integrated to solve for the motion of the part due to the impact. To import these data into Blender, the initial reference coordinates of the nodes and element connectivity matrix were extracted from the finite element model, as well as the mode shapes and the integrated modal degrees of freedom which were to be used to define vertex displacements in the simulator. All data were written to external files which could be loaded through Blender's Python programming interface, allowing the mesh import to be automated.

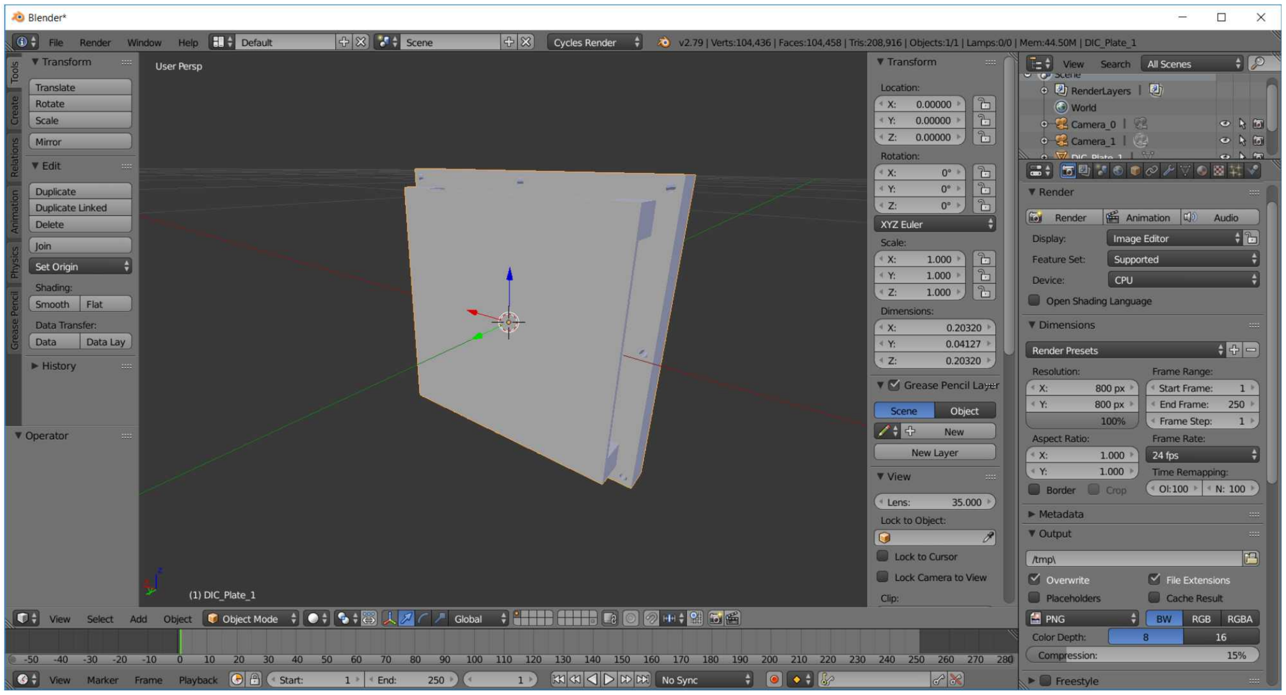
Note that because Blender utilizes surface meshes for its geometry rather than the volume meshes used by many 3D finite element codes, any volumetric finite element models should first be skinned prior to importing into Blender. While this is not a functionality included in many finite element packages, it can be performed by looping through all faces of all elements in the finite element model and only keeping those faces that occur in a single element, as interior faces will appear in two elements.

3.2 Loading the Mesh into Blender

To begin, an empty scene was created in Blender. The renderer was also switched from the default Internal renderer to the Cycles renderer to create more photo-realistic images. The plate structure was created in Blender with the extracted finite element coordinates and connectivity matrix using the Python programming interface. Upon successful creation of the mesh in Blender, it should be visible in the 3D View, as shown in Figure 3.

3.3 Applying Materials and Textures to the Mesh

The newly created plate Object will not have any texture applied to the surface, and will instead be rendered with a default gray Material. In order to perform a physical DIC experiment, a speckle pattern is typically applied to the part. We follow the same


Blender*

v2.79 | Verts:104,436 | Faces:104,458 | Tris:208,916 | Objects:1/1 | Lamps:0/0 | Mem:44.50M | DIC_Plate_1

Transform
Location: X: 0.00000 Y: 0.00000 Z: 0.00000
Rotation: X: 0° Y: 0° Z: 0°
XYZ Euler
Scale: X: 1.000 Y: 1.000 Z: 1.000
Dimensions: X: 0.20320 Y: 0.04127 Z: 0.20320

View
Lens: 35.000

Render
Display: Image Editor
Feature Set: Supported
Device: CPU

Dimensions
Resolution: X: 800 px Y: 800 px 100%
Frame Range: Start Frame: 1 End Frame: 250 Frame Step: 1
Aspect Ratio: X: 1.000 Y: 1.000
Frame Rate: 24 fps
Time Remapping: Ol:100 N: 100

Output
/tmp\
Overwrite ✓ File Extensions ✓
PNG BW
Color Depth: 8
Compression: 15%

Start: 1 End: 250 1 No Sync

Figure 3: Plate mesh loaded into Blender

approach here. The user may create an image of a speckle pattern, or a picture of an actual test setup may be applied to the mesh (see Section 6). In this example, a 6400 x 6400 pixel white image was created, and 20 pixel radius black speckles were placed randomly over the image, creating the image shown in Figure 4. Within Blender, a new material was created using a Diffuse surface shader, and an image texture was applied using this image.

Initially, the material does not appear to be applied to the object; this is because the projection between the 3D mesh and the 2D image has not yet been defined. Blender needs to be instructed how to unwrap the mesh so that it can apply the speckle image to it. In Blender, it is possible to select edges to use as seams during unwrapping, as well as to project the mesh to specific views (for example as shown in Section 6); however, for this case we will use a simple cube projection to unwrap the mesh. After the mesh is unwrapped, the mesh vertices can be positioned on the image texture using the UV/Image Editor, and the Mesh should show the texture applied in the 3D view, as shown in Figure 5. An additional metal Material was created using a combination of Diffuse and Glossy shaders; this material was applied to portions of the Mesh that did not have the speckle pattern applied.

3.4 Adding Cameras, Lights, and Adjusting the Speckle pattern

For any DIC setup, cameras will need to be intelligently placed. In this case, two cameras are added, pointed at the center of the plate and rotated approximately 30 degrees with respect to one another. The camera lenses were specified as 95 mm focal length within the Blender software so the speckled portion of the plate filled the image. While the relative angle between the cameras may be considered wide given the focal length of the lenses (see e.g. [10]), it was thought to be acceptable due to the primarily out-of-plane motion expected from the part. The aperture was set to $f/8$ to simulate a finite depth of field. At this aperture, only the very edges of the speckled plate have begun to blur, so this depth of field would be considered adequate for this test. A spotlight lamp was also added to illuminate the part. At this point, our scene appears as shown in Figure 6. Note that the ability to synthetically predict image effects due to lens focal length, aperture, and camera sensor size is a valuable application of Blender; a test engineer can identify which lenses and cameras would be most effective in performing a given test prior to any actual test setup.

At this point the scene can be rendered for the first time. A histogram can be plotted to help in analyzing the render. In our case, the image has rendered severely overexposed when using the default parameters for the spotlight lamp (Figure 7a). We can adjust this by reducing the intensity of the light in the scene (Figure 7b). This is also the point where the speckle pattern can be scaled to achieve a proper pixel-per-speckle ratio.

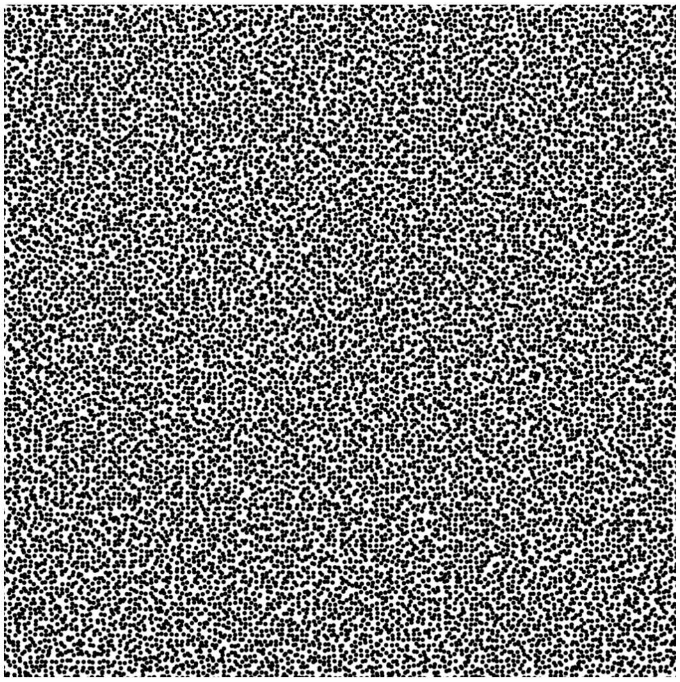
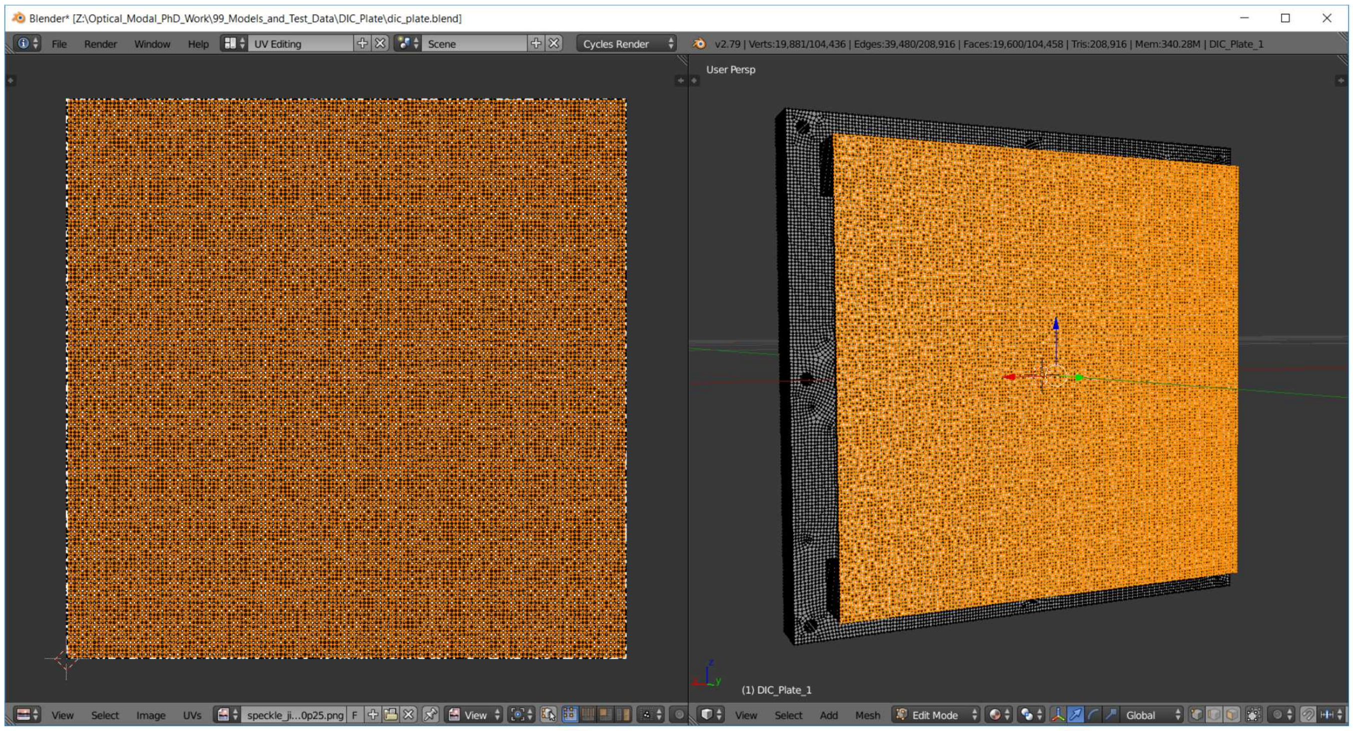


Figure 4: Speckle pattern applied to the part.



Blender* [Z:\Optical_Modal_PhD_Work\99_Models_and_Test_Data\DIC_Plate\dic_plate.blend]
File
Render
Window
Help
UV Editing
Scene
Cycles Render
v2.79 | Verts:19,881/104,436 | Edges:39,480/208,916 | Faces:19,600/104,458 | Tris:208,916 | Mem:340.28M | DIC_Plate_1
User Persp
(1) DIC_Plate_1
View
Select
Image
UVs
speckle_ji...0p25.png
View
Add
Mesh
Edit Mode
Global

Figure 5: UV/Image Editor showing the unwrapped mesh (orange) superimposed on the image texture (left), and the image texture projected to the mesh in the 3D view (right).

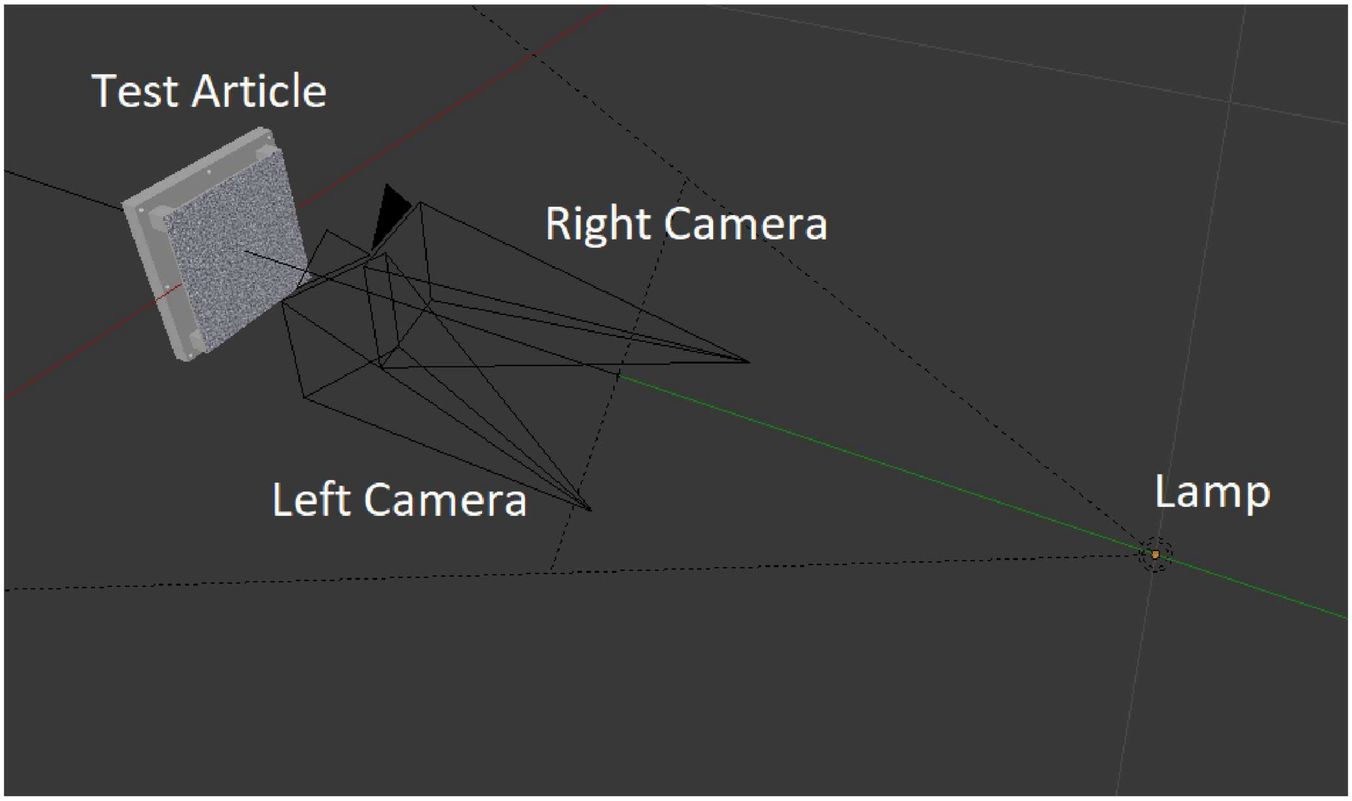


Figure 6: Scene with 95mm focal length cameras and a spotlight.

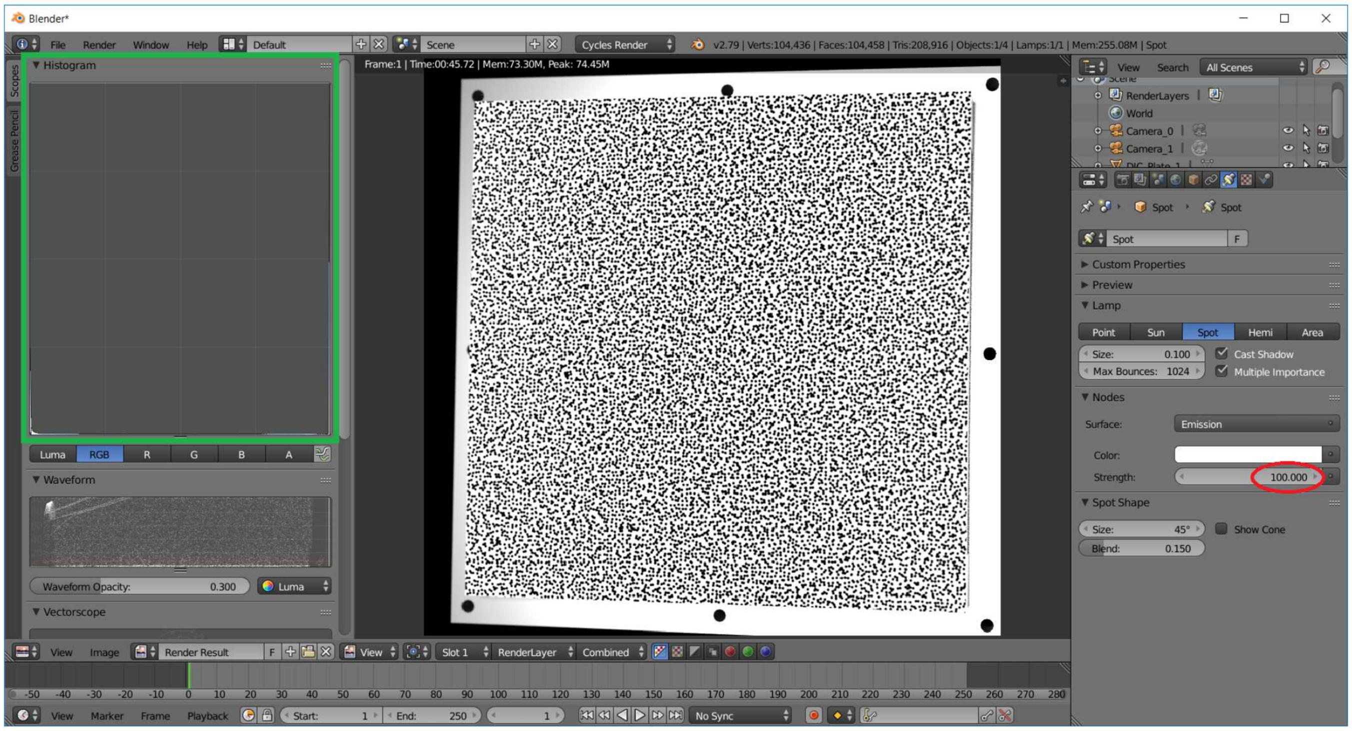
3.5 Rendering a Series of Images

While it would be possible to deform the mesh and render an image using the graphical interface, it would become tedious to render a series of images this way. Instead, the programming interface is used to automate the deformation of the mesh and render an image from each camera at each time step. Since the modal coordinates **q** were integrated to produce a time history, they need to be multiplied by the mode shape matrix Φ to produce physical displacements **x**.

$$\mathbf{x} = \Phi \mathbf{q} \tag{1}$$

The displacements at each time step were added to the initial finite element coordinates of each vertex in the Mesh to produce the new coordinate at which the vertex would be placed for that time step. After the vertex positions were updated, the software rendered an image for each camera.

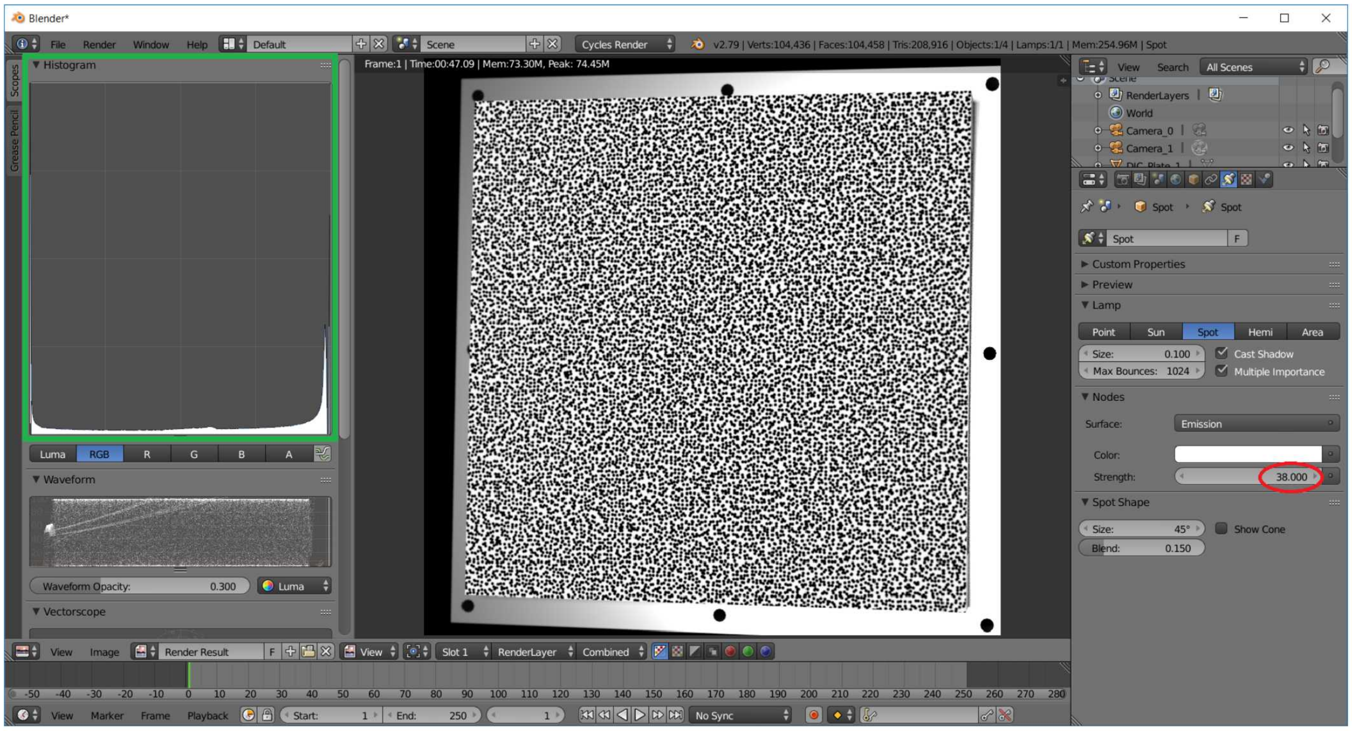
Figure 8 shows an example render for the left and right camera images. Figure 9 shows the difference in pixel intensities at the peak displacement image compared to the reference image; the maximum difference is only about 20 gray-levels (out of 256 for an 8-bit image).


v2.79 | Verts:104,436 | Faces:104,458 | Tris:208,916 | Objects:1/4 | Lamps:1/1 | Mem:255.08M | Spot

Frame:1 | Time:00:45.72 | Mem:73.30M, Peak: 74.45M

Lamp: Spot
Size: 0.100
Max Bounces: 1024
Cast Shadow ✓
Multiple Importance ✓
Surface: Emission
Strength: 100.000
Spot Shape Size: 45°
Blend: 0.150
Waveform Opacity: 0.300
Start: 1
End: 250

(a) Initial render showing severe overexposure. The histogram shows many pixels that are clipped in the top brightness bin.

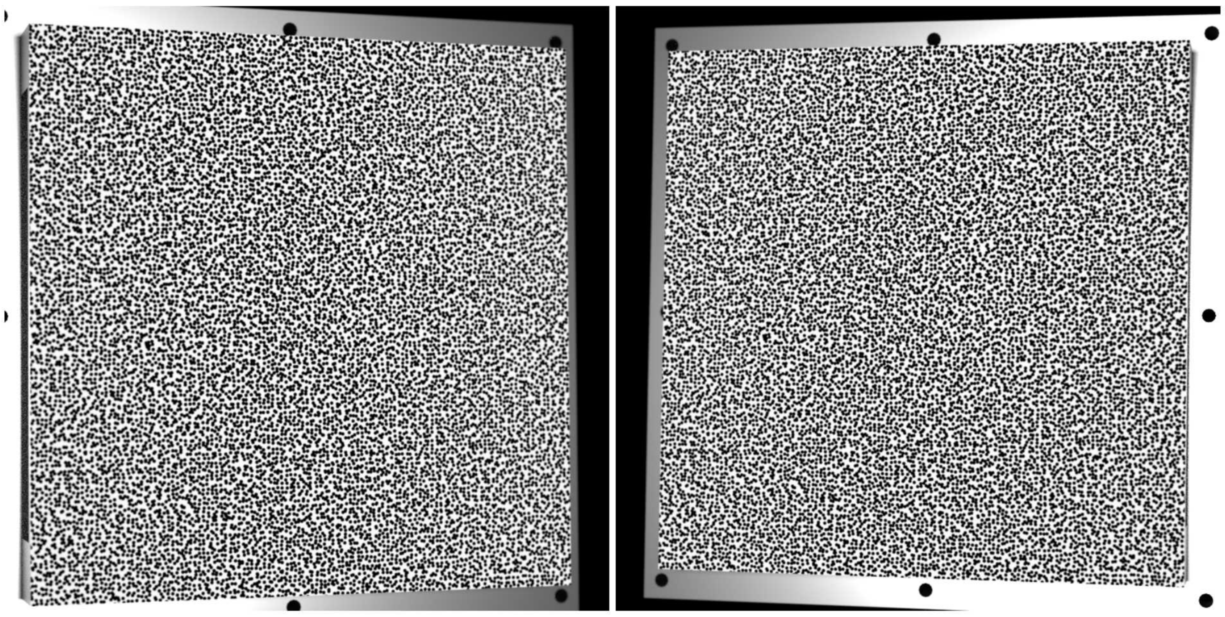

v2.79 | Verts:104,436 | Faces:104,458 | Tris:208,916 | Objects:1/4 | Lamps:1/1 | Mem:254.96M | Spot

Frame:1 | Time:00:47.09 | Mem:73.30M, Peak: 74.45M

Lamp: Spot
Size: 0.100
Max Bounces: 1024
Cast Shadow ✓
Multiple Importance ✓
Surface: Emission
Strength: 38.000
Spot Shape Size: 45°
Blend: 0.150
Waveform Opacity: 0.300
Start: 1
End: 250

(b) Initial render showing proper exposure. The histogram shows only a few clipped pixels, and these are in the specular reflection on the semi-glossy metal back plate of the model.

Figure 7: Two renders with varying illumination. Note the differences in the histogram (boxed in upper-left corner of the image) due to the changing of the lamp strength (circled on right-hand side of the image)



(a) Left Camera Render **(b)** Right Camera Render

Figure 8: Example renders of left and right camera images

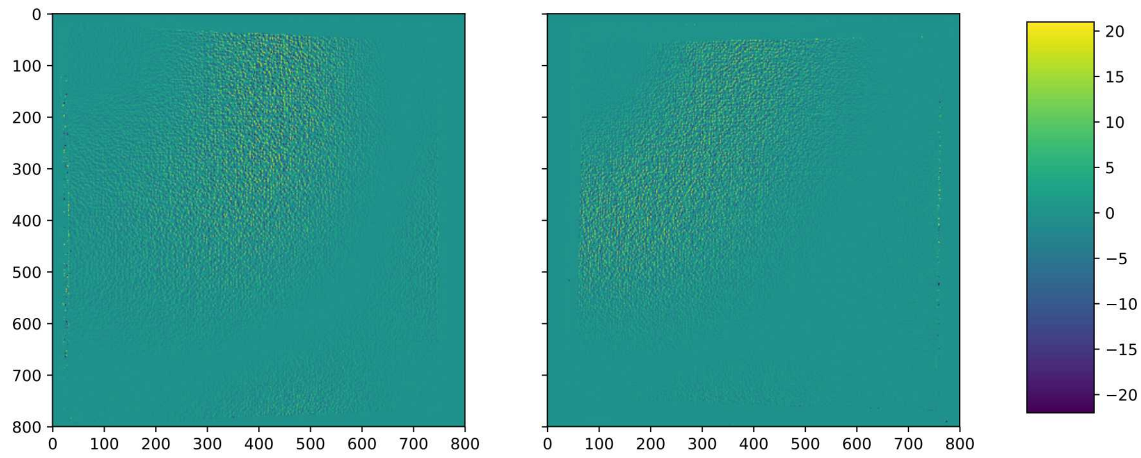


Figure 9: Image difference between the reference image and the image containing the maximum displacement.

4 Digital Image Correlation Example Analysis

To demonstrate that the synthetic images from the simulator can be used for DIC applications, the workflow images were analyzed using the open-source Digital Image Correlation Engine (DICe) software [11].

4.1 Camera Calibration

DIC software typically requires a camera calibration step that effectively teaches each camera where it is in space with respect to the other camera, as well as any intrinsic camera lens parameters. This allows a given feature or subset on the image pair to be triangulated to determine its location in space. Two options are available in the optical test simulator.

The first follows the traditional DIC test: with the cameras in the same places as they would be during the test, take a series of images of a calibration target that can be fed into the analysis to compute the camera calibration. To perform the traditional calibration, a model of a calibration target was created using a rectangular Mesh textured using a 10 mm-spaced calibration target texture[4]. The textured Mesh with 10 mm grid spacing is shown in Figure 10. Again, the calibration target could be moved through the camera field of view manually using the graphical interface and rendered at each position, but since a reasonably large number of calibration images were to be rendered, a Python script was instead written to automate the translation and rotation of the calibration target and rendering of the images. The script specified three positions in each of the three directions for 27 total positions in the field of view. At each position, a 10-degree rotation about each axis was performed (both in positive and negative directions) and an image was rendered with each camera for a total of 7 rotations at each position (which includes an image of the unrotated calibration target). This resulted in 189 pairs of calibration images; Figure 11 shows an example stereo-pair of these synthetic calibration images. These images can be fed into a DIC package to perform the calibration.

A second approach that is available when using synthetic images is to directly populate the calibration parameters for each camera in the DIC software, which can be derived from the camera matrices $[K]$ and $[R|T]$. This ensures there are no errors due to the calibration process. The $[K]$ and $[R|T]$ matrices,

$$[K] = \begin{bmatrix} \frac{p_u f_{mm}}{ss_{u,mm}} & s & (1/2 - s_u)p_u \\ 0 & \frac{p_v f_{mm}}{ss_{v,mm}} & (1/2 - s_v)p_v \\ 0 & 0 & 1 \end{bmatrix} \tag{2}$$

$$[R|T] = \left[\begin{array}{ccc|c} r_{xx} & r_{xy} & r_{xz} & t_x \\ r_{yx} & r_{yy} & r_{yz} & t_y \\ r_{zx} & r_{zy} & r_{zz} & t_z \end{array}\right] \tag{3}$$

represent the intrinsic and extrinsic camera parameters respectively, and transform the physical coordinates xyz to image coordinates uv

$$w \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = [K][R|T] \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{4}$$

Note that equation (4) is constructed in homogeneous coordinates, resulting pixel locations on the left-hand side of the equation will be scaled by the third row of the vector, w. Dividing all terms in the vector by the third term w will result in the true pixel locations uv being recovered.

In the $[K]$ matrix, which specifies each camera's intrinsic properties, p_u and p_v are the image resolution in pixels in the u and v dimensions, respectively, $ss_{u,mm}$ and $ss_{v,mm}$ are the camera sensor size in millimeters in the u and v dimensions, respectively, s_u and s_v are the normalized camera shift values in fraction of the image in the u and v dimensions, respectively, and f_{mm} is the lens focal length in millimeters. All of these camera parameters are specified for each camera in the Blender software, so the $[K]$ matrix can be recovered directly. Some camera calibration routines will return a camera skew s or a separate focal length f_{mm} for each of the u and v directions; however Blender's camera model cannot handle these parameters. If a significant skew or difference in focal length between dimensions is needed for the camera and lens combination used in the test, these distortions can be applied via postprocessing the output images from Blender. The extrinsic matrix $[R|T]$ can be populated from each

[4]Calibration target textures can be generated using the Calibration Target Generator, which is available for download from the Correlated Solutions website `https://www.correlatedsolutions.com/software-downloads/`

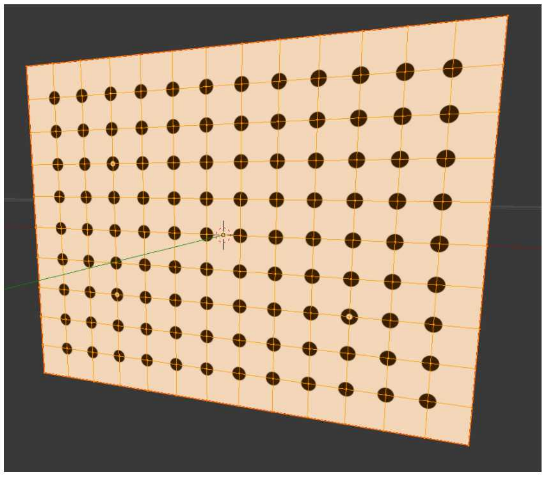


Figure 10: Calibration target model showing mesh grid with 10 mm spacing.

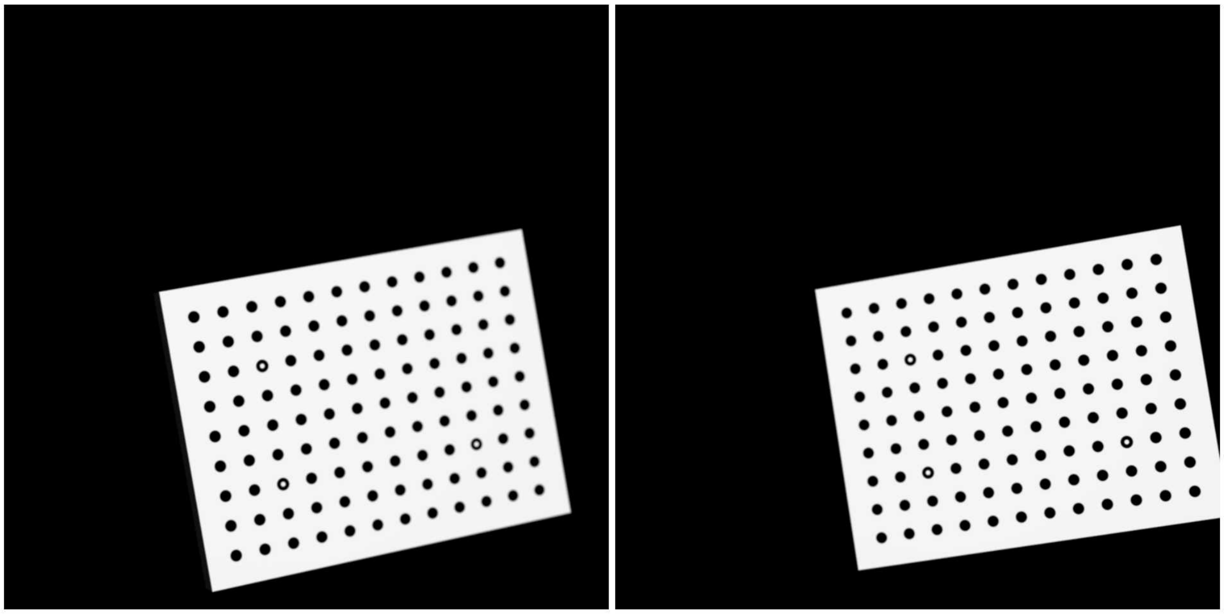


Figure 11: Example Synthetic Calibration Target Images

camera object's location and orientation, and is the homogeneous transformation between the global and camera coordinate systems. Since Blender uses a perfect pinhole camera model, no distortion exists in the lens, so all distortion parameters can be set to zero.

Importing the calibration parameters into the DIC software will be specific to the software package being used, but for DICe, the rotation matrix must be transformed to Cardan-Bryant (x,y,z) angles, and the translation vector is specified as the vector from the camera origin to the global coordinate system origin in the camera coordinate system. These data are packaged into an XML file format specified by the DICe documentation which can be imported into the DICe software.

4.2 DIC Analysis

After the camera calibration was performed with the synthetic calibration images, the deformed images were loaded into the software. DIC was performed using a subset of 41 pixels with a step of 21 pixels; each subset contained approximately 20-30 speckles, which is larger than typically recommended by standard of practice documents (e.g. [10]). However, in structural dynamics applications where motions tend to be small and deformation shapes tend to be smooth, it is often acceptable to reduce the spatial resolution of the measurement to decrease the measurement noise floor. Figure 12 shows the subset displacement at the center of the plate reached a maximum of approximately 1/10th of a pixel and decayed from there. Figure 13 shows a comparison of the 3D displacement prescribed at the center of the plate and the displacement measured by the DIC analysis. The out-of-plane motion of the plate was measured very accurately. The in-plane motions, being very small, were not measured as accurately. Figure 14 shows exaggerated 3D deflection shape extracted from the DIC analysis at peak deflection of approximately 1/10 of a millimeter.

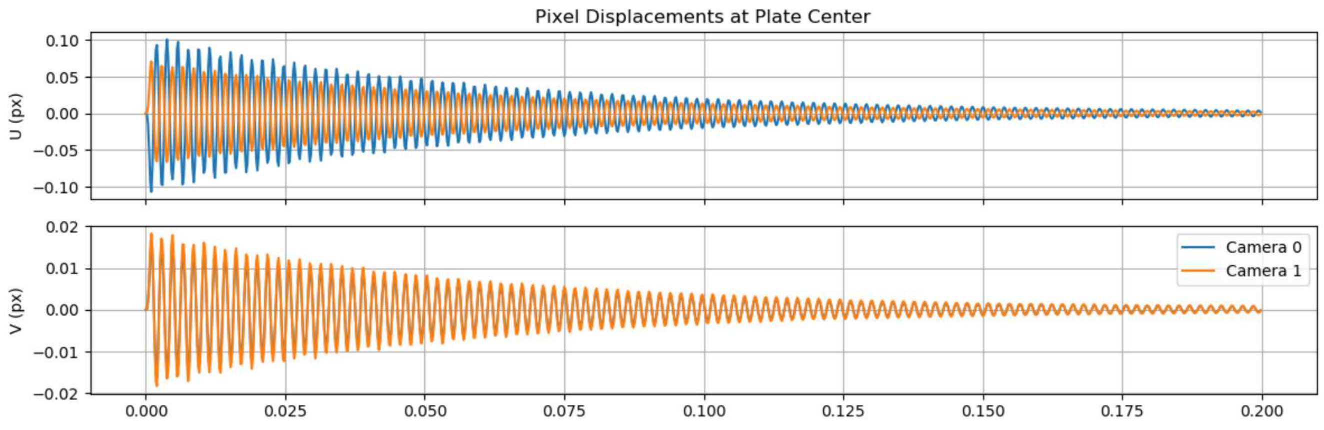


Figure 12: Subset motions in pixels at the center of the plate. Peak displacement of approximately 0.1 pixels was measured.

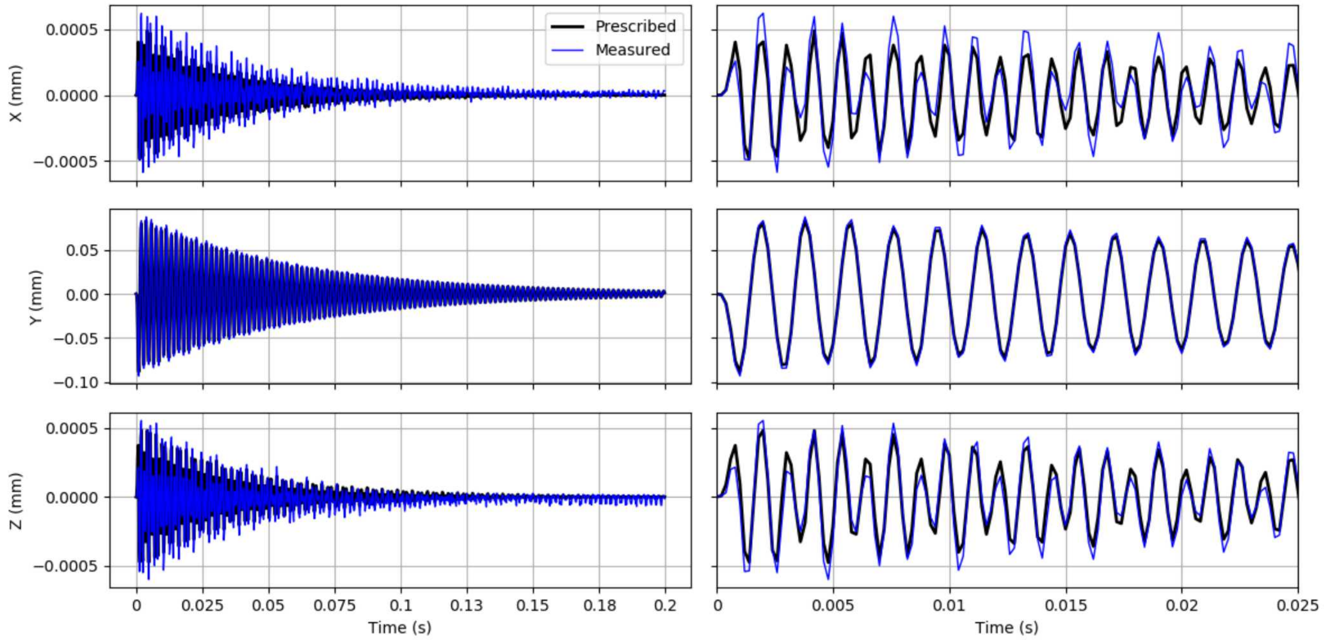


Figure 13: Comparison between prescribed displacement on the Blender mesh and displacement measured by the DIC analysis. The *y*-direction (out-of-plane) response is measured accurately, while the in-plane responses, being very small, are not measured as accurately (A coordinate system is shown in Figure 2). The right figures show a zoom of the first 0.025 seconds of the time history.

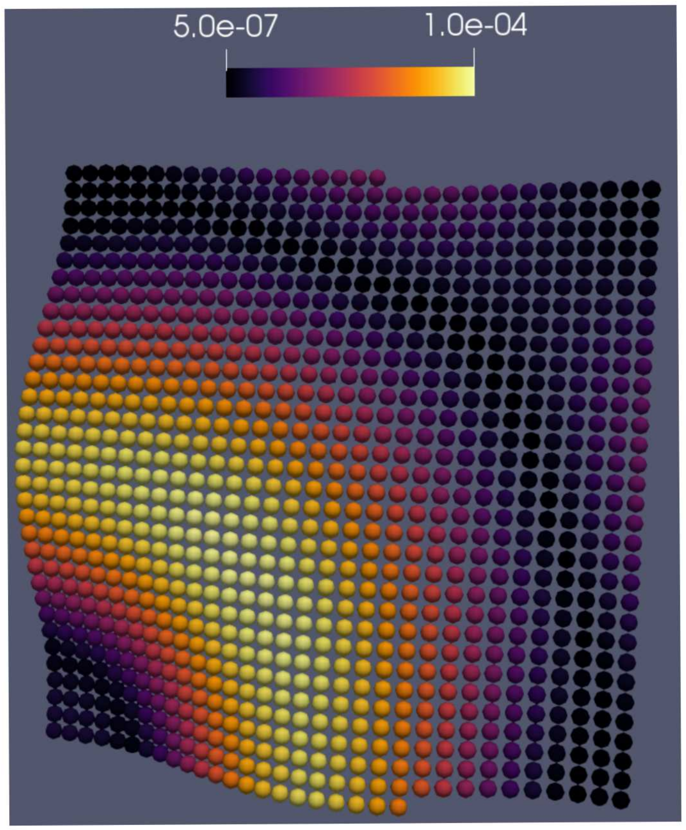


Figure 14: Displacement shape at peak deflection of the plate.

5 Sub-pixel Resolution Analysis

The example workflow showed that Blender could accurately render sub-pixel motions on the image; however a more rigorous study was desired. As Blender was not designed as an engineering tool, and instead was designed for computer graphics applications, it was important to determine how accurately the software could produce images with sub-pixel displacements. DIC, at least in structural dynamics applications where displacements can be very small, often relies on the ability of the algorithm to extract sub-pixel motions of the unit under test. Therefore, if the Blender software cannot accurately produce such images, its utility as a synthetic image generator for structural dynamic applications is greatly reduced.

To help understand the ability to resolve subpixel displacements using the Blender software, the subpixel rigid body translation analysis from [4, 12] was reproduced using synthetic Blender images. Both Blender render engines were investigated for this analysis. The DICe software was again used to measure the displacements, and these were compared to the prescribed displacements in the optical simulator. Note that this analysis does not separate the measurement errors of the DIC approach from rendering errors in the simulated images; the desire is instead to show that error distributions from DIC analysis of Blender's synthetic images are characteristic of those obtained from DIC analysis, which would imply that the synthetic images are not adding any significant errors to the analysis.

A simple flat plate mesh was created and textured with a 6400 x 6400 pixel speckle texture. Unidirectional, rigid displacements were applied in the Blender software from 0 to 2 pixels in 41 steps. A 512 x 512 pixel image was rendered at each displacement level, resulting in an approximate factor of 10 downsampling from the speckle image texture to the final image resolution. The final speckles had a diameter of approximately 5-6 pixels. A 41 x 41 pixel subset was used to compute displacements, with a step size of 21, for 441 subsets across each image. Figure 15 shows the reference image and an example subset used in the DIC analysis. Pixel displacements for each subset were compared against the prescribed value to compute an error value. Error mean and standard deviation were computed over all subsets at each displacement level. This analysis was repeated for each set of rendering parameters analyzed.

For the Internal render engine, oversampling within a given render is specified by an antialiasing parameter. Discrete options

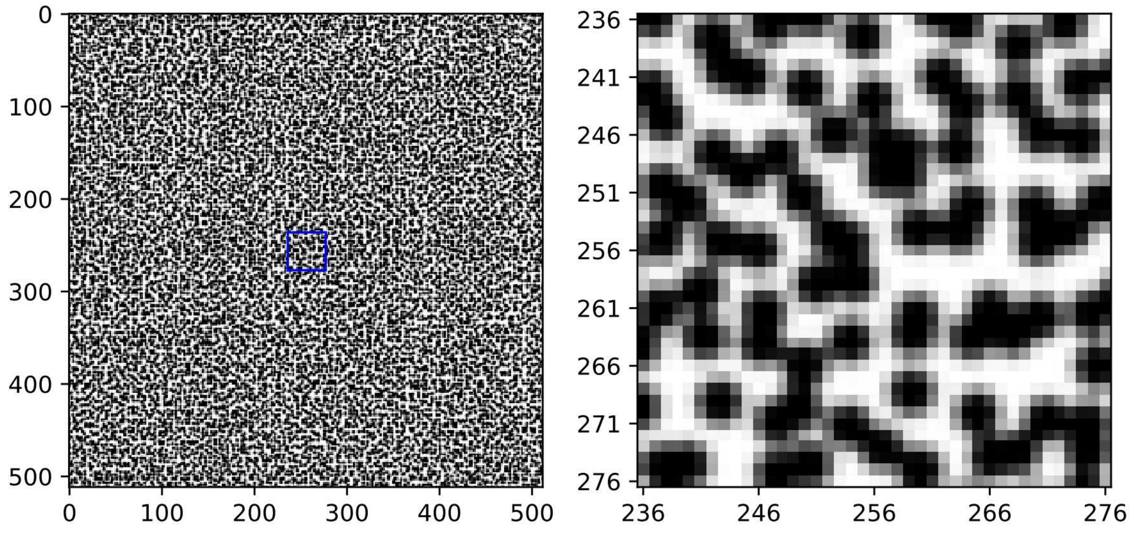


Figure 15: Image and subset used for subpixel displacement analysis.

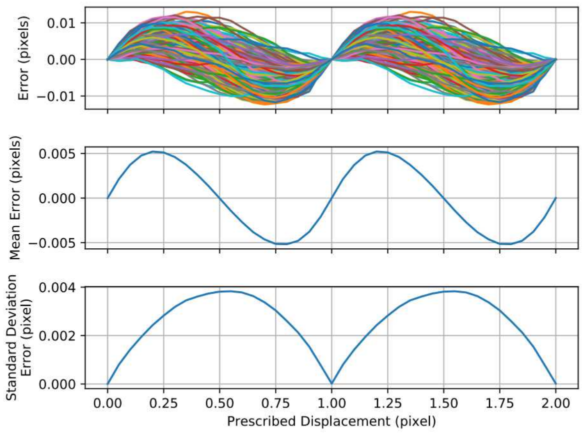
of 5, 8, 11, and 16 samples per pixel are available[5]. Since the antialiasing options in the internal engine are somewhat limited, an additional oversampling step was imposed which rendered the image at a higher resolution and then down-sampled back to the desired resolution using a bi-cubic interpolation. For this separate oversample operation, factors of 1, 2, 5, 10, and 25 were examined. Figure 16 shows the displacement errors for each subset over 6 of the 20 sets of render parameters analyzed. Figure 17 shows the peak error and standard deviation over all antialiasing and oversampling parameters.

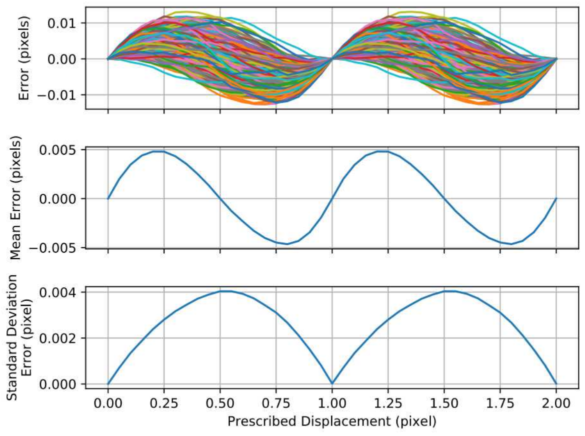
The Internal render engine, which is deterministic, perfectly reproduces the reference image at integer pixel displacements; all subset errors go to zero. It appears that the subpixel accuracy is not heavily dependent on the antialiasing parameter in the Blender software, but does depend on the amount of oversampling used. Rendering at a higher resolution then down-sampling appears to increase the peak error, but also reduces the variability in the subset error. The minimum variance in the subset displacement error appears to occur with an oversample factor of about 2, and increasing the oversampling more than that only appears to increase the variance of the subset errors.

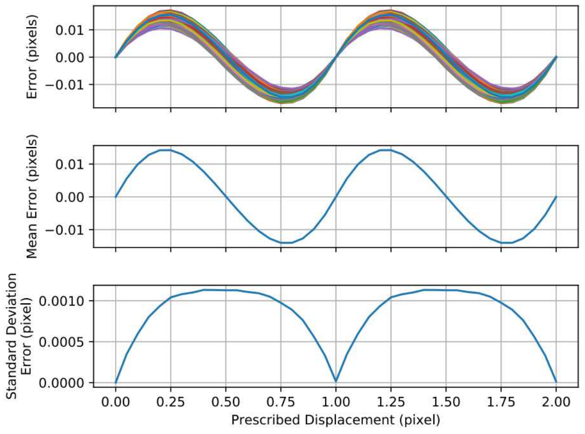
A similar analysis was performed on the Cycles render engine. The Cycles render engine is non-deterministic in that it passes a specified number of samples for each pixel, and those samples each traverse a random path through the scene. For example, in a glossy, transparent material, some light is transmitted through the material and some light is reflected off the material. In the Cycles render engine, a given sample would randomly traverse one of those paths, with the probability of reflecting or transmitting determined by the material model used on the mesh. Only when a large number of samples are cast and the resultant pixel intensities averaged does the material begin to look realistic. This is also how the Cycles renderer handles antialiasing; if there is a pixel in which a sharp speckle boundary occurs, some of the rays cast into the pixel will return black and others will return white, resulting in a gray final pixel intensity. The percentage of rays returning black will depend on the percentage of the pixel that is covered by a black texture. If a render is insufficiently sampled, the image will appear noisy. This is especially prevalent in indirect lighting scenarios (e.g. in shadows) where the primary source of lighting on a surface is due to reflections of light off of other surfaces. An insufficient number of samples can also result in the antialiasing quality being reduced, which is especially problematic for algorithms such as DIC which rely on accurate gray-values at speckle boundaries to perform subpixel interpolation.

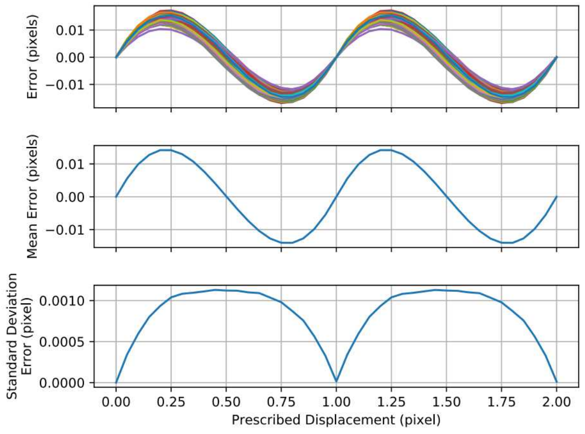
For this analysis, the number of samples per pixel used to render the image was varied from 4 to 8192. Figure 18 shows subset displacement errors and statistics for four of the sampling parameters. Figure 19 shows the peak error and standard deviation over all sampling parameters. It can be seen that when the render is insufficiently sampled, for example in Figure 18a, the image

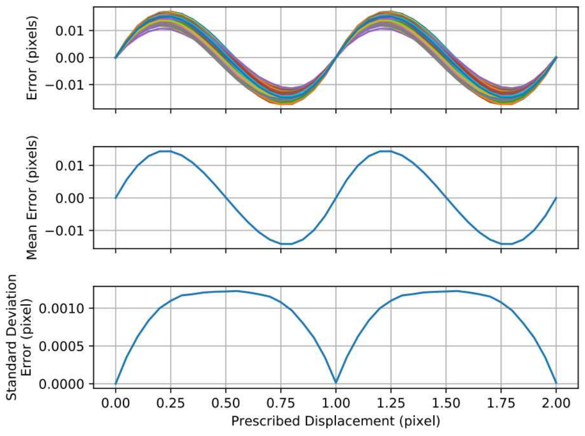
[5]See `https://docs.blender.org/manual/en/2.79/render/blender_render/settings/antialiasing.html` for more information


(a) Antialiasing: 5 samples per pixel
Oversampling: 1x


(b) Antialiasing: 16 samples per pixel
Oversampling: 1x


(c) Antialiasing: 5 samples per pixel
Oversampling: 5x


(d) Antialiasing: 16 samples per pixel
Oversampling: 5x


(e) Antialiasing: 5 samples per pixel
Oversampling: 25x

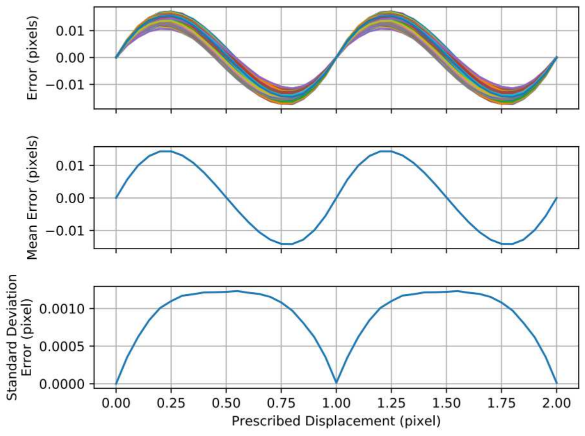
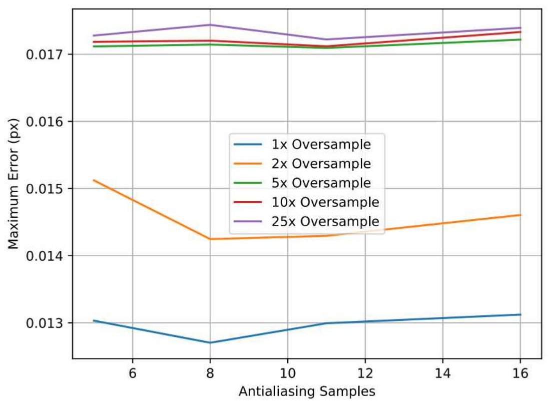
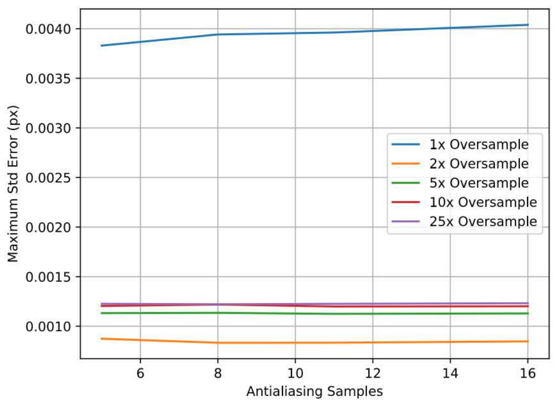

(f) Antialiasing: 16 samples per pixel
Oversampling: 25x

Figure 16: Internal render engine errors and statistics.



(a) Peak displacement error over all subsets



(b) Standard deviation of displacement error over all subsets.

Figure 17: All statistics for the internal renderer.

noise inherent in the render technique results in distortion of the typical sinusoid-like interpolation error curve; where the errors would typically diminish at integer pixel displacements, they instead continue to grow. As the render sampling is increased, the effective image noise decreases, and the interpolation error curve looks like the typical function again. The curves in Figure 19 show that by approximately 1000 samples, the images have converged, and the maximum and standard deviation of the subset errors stop decreasing. It should be noted that this test scene was a single surface that was directly lit with a spotlight lamp, so this image will be relatively noise-free as far as the Cycles renderer is concerned; if the generated synthetic images have shadows, reflections, or indirect lighting, it may be necessary to further increase the number of samples from the values found here to create sufficiently accurate renders.

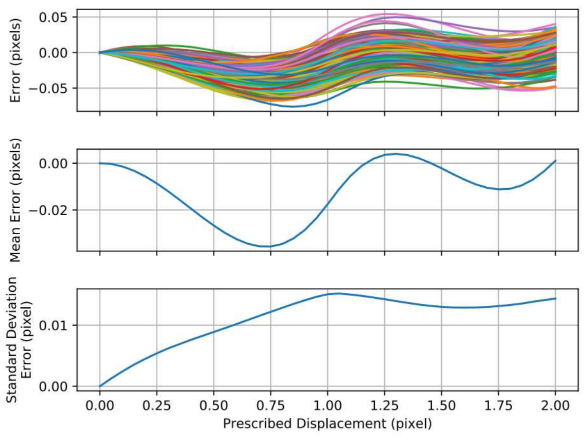
The Internal and Cycles render engines produce images with similar subpixel trends. Both render engines produce images that when analyzed using the DICe software have pixel errors on the order of 0.01 pixels. The Cycles engine seems to produce less variance in the errors than the Internal engine, except at integer pixel displacements. At integer pixel motions, the Internal renderer produces identical images, just shifted by a pixel. This can be seen by both the mean and standard deviation of the displacement errors for all subsets going to zero. For the Cycles renderer the mean displacement error will go to zero. The variance of the errors, however, does reach zero at integer pixel locations, as there is noise from the render technique on the image. The variance does approach zero as render samples are added and noise is reduced. The errors in Figures 16 and 18 have similar shape and magnitude to those found in [4, 12], suggesting that the synthetic images rendered by Blender are not introducing any additional error to the DIC analysis.

6 Test Planning and Simulation: Example Workflow

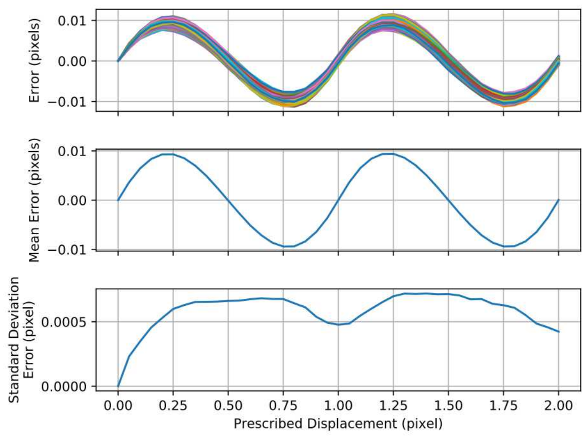
One application for which this optical test simulator has been used successfully is test planning. A test was to be performed on the Box Assembly with Removable Component (BARC) structure [13] shown in Figure 20, and the test engineers were interested if the responses could be measured using DIC.

The BARC finite element model geometry was loaded into the Blender software and cameras were placed approximating the desired view. Camera matrices were extracted from the Blender cameras, allowing the test planners to get an approximate projection from 3D space to the 2D image. A pseudo-random excitation was applied to the finite element model to predict displacements and those displacements were run through the camera matrices using equation (4) to estimate the displacement in pixels that could be expected from the test. This application of the optical test simulator can result in large time savings during testing; it can take a long time to download and process test images to extract displacements, so needing to repeat the test can be a large penalty if the excitation level of a test was too small to produce visible deflections.

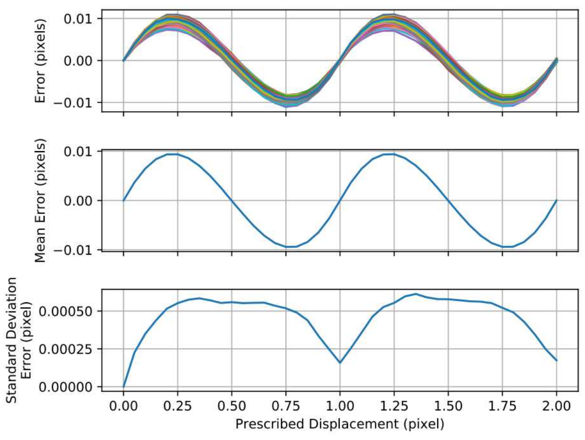
The optical test simulator also allows testing of speckle patterns to optimize spot size. Rough calculations can of course be performed to estimate the speckle size required for a given resolution, but the Blender software makes it easy to iterate and



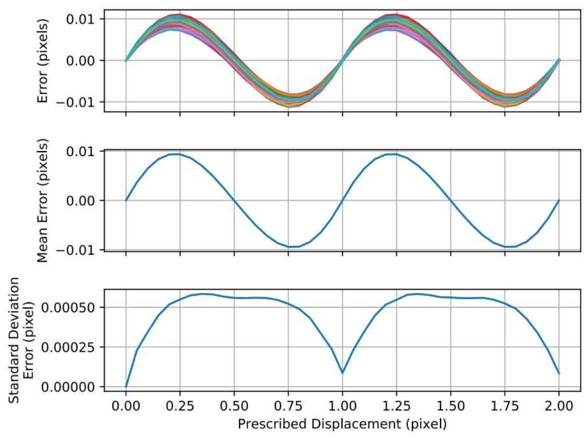
(a) 4 samples



(b) 512 samples

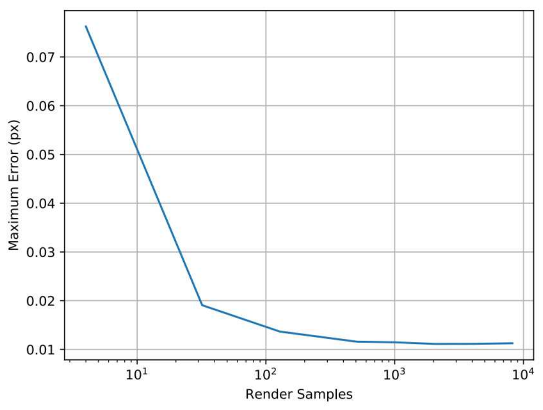


(c) 2048 samples

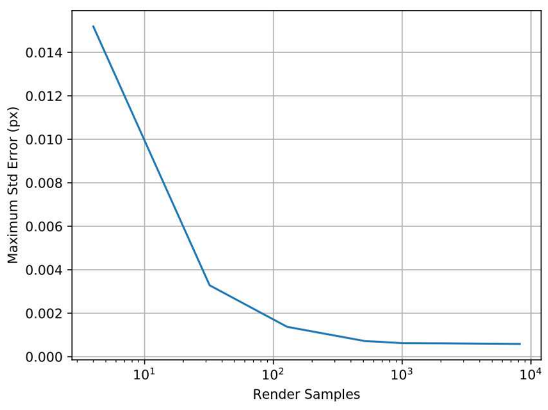


(d) 8192 samples

Figure 18: Cycles render engine errors and statistics.



(a) Peak displacement error over all subsets



(b) Standard deviation of displacement error over all subsets.

Figure 19: All statistics for the Cycles renderer.

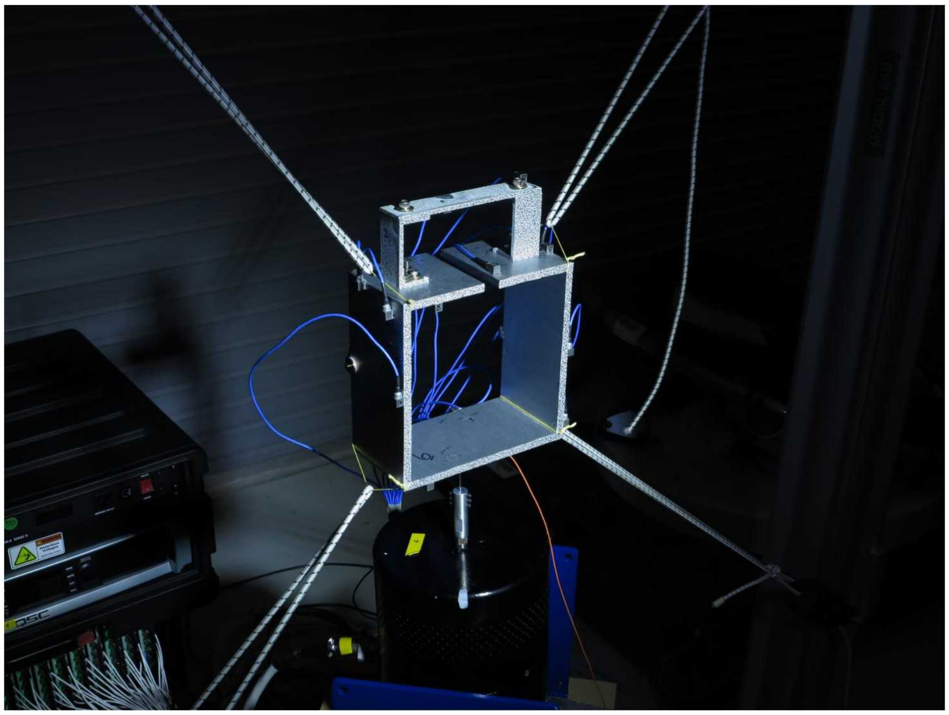


Figure 20: BARC optical test to be replicated using the optical test simulator.

visualize what the speckle pattern will look like. In this case, a speckle pattern was loaded into Blender and mapped to the surfaces of interest as described in Section 3.3. Then, the relative scale of the image mapping coordinates in the Blender UV/Image Editor was modified until there were approximately 3-4 pixels per speckle. Blender has the ability to export the image mapping coordinates, so when an adequate scaling was achieved the image coordinates were exported to an image processing application (e.g. Photoshop or GIMP), and the image coordinates were used as a template to extract a portion of the speckle pattern that was applied in this simulator, shown in Figure 21. The template shown in Figure 21 was then printed onto label paper and directly adhered to the test article. Using this technique, the experiment can use the exact speckle pattern that was validated using the optical test simulator.

Once the speckle pattern is set and the test article prepared for test, the cameras in the experiment were set up. While the initial Blender cameras were a reasonable approximation for the final experimental set up, it would obviously be very difficult and time consuming to try to force the experiment cameras to exactly match the Blender cameras, especially when the Blender cameras are much easier to reposition in-software. With the cameras set up, a camera calibration was performed. From this calibration, intrinsic and extrinsic camera parameters were extracted; however, the extrinsic parameters were defined in a camera rig coordinate system. Therefore, the goal was to determine the rigid transformation from the rig coordinate system to the part coordinate system in which the finite element model and Blender scene were defined.

To solve for this transformation, a set of points needed to be defined in both the rig and part coordinate systems. Six node locations corresponding to the visible outside corners of the box and attached component were extracted from the finite element model to create the list of points in the part coordinate system. To select points in the camera rig coordinate system, 3D coordinates of these same points needed to be triangulated from image coordinates; this was performed using the `triangulatePoints` function of the OpenCV [14] library. This function accepts projection matrices $[P] = [K][R|T]$ as well as a list of pixel coordinates for each camera and returns a set of 3D coordinates corresponding to those pixel locations. With a set of points in 3D space for each coordinate system, the transformation between the two was computed using an singular-value-decomposition-based least-squares rigid transformation solver [15, 16]. These original extrinsic camera matrices were post-multiplied by this transformation to produce extrinsic matrices defined in the finite element model coordinate system. The Blender cameras were reproduced in the scene using the intrinsic matrix from the calibration and the updated extrinsic matrix. The view in Blender from the left reconstructed camera is shown in Figure 22.

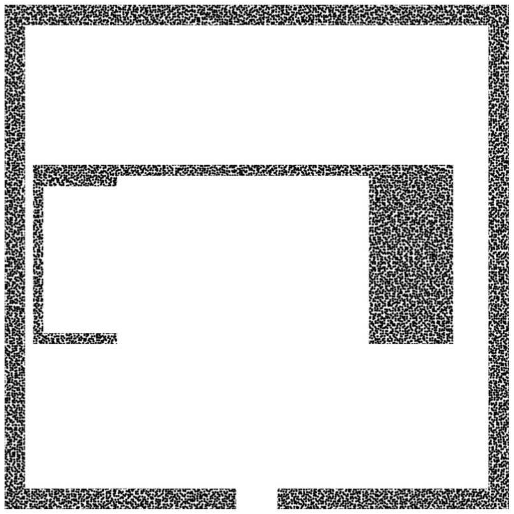


Figure 21: Speckle pattern imposed onto image coordinates from Blender to create speckle templates that could be directly applied to the structure.

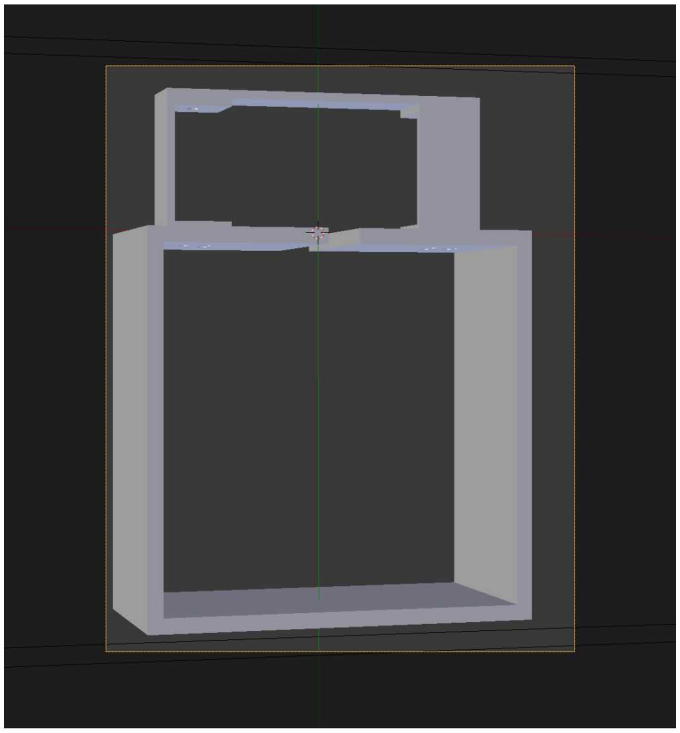


Figure 22: View of the finite element model from the reconstructed camera.

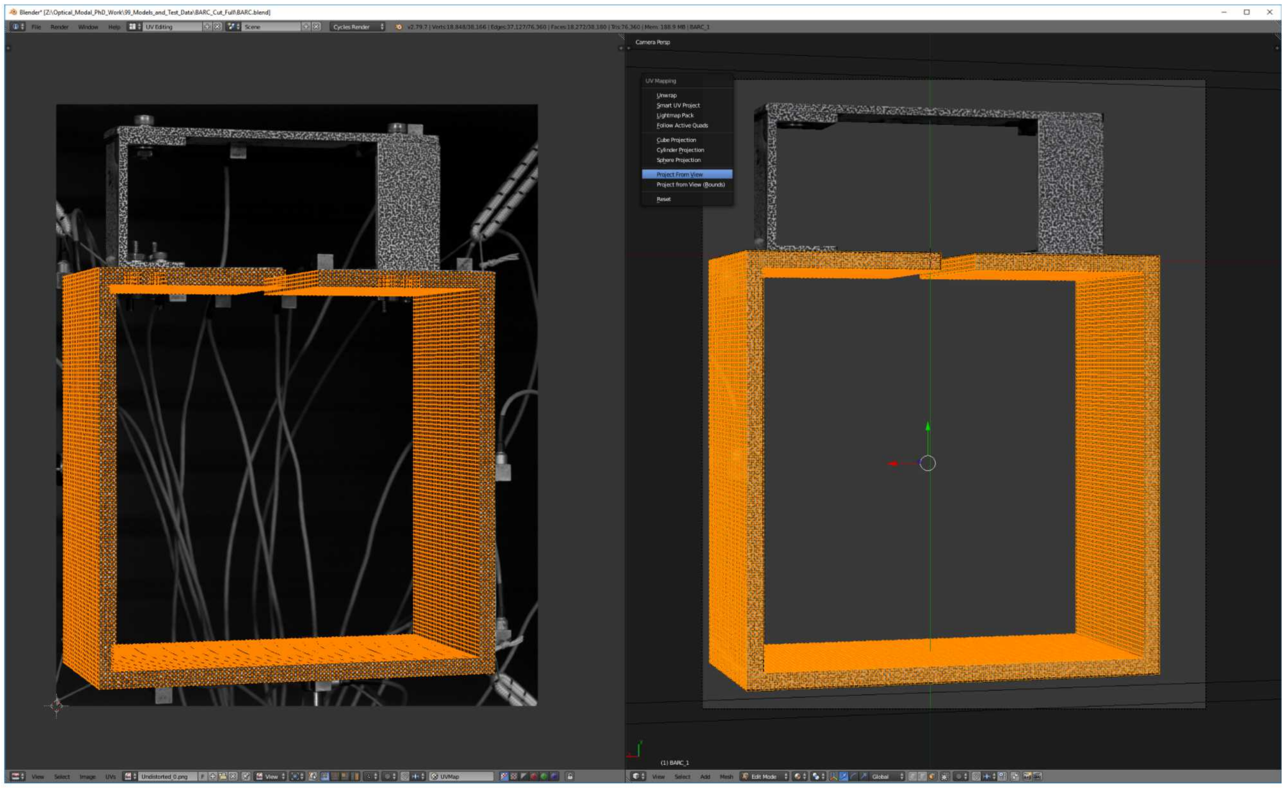
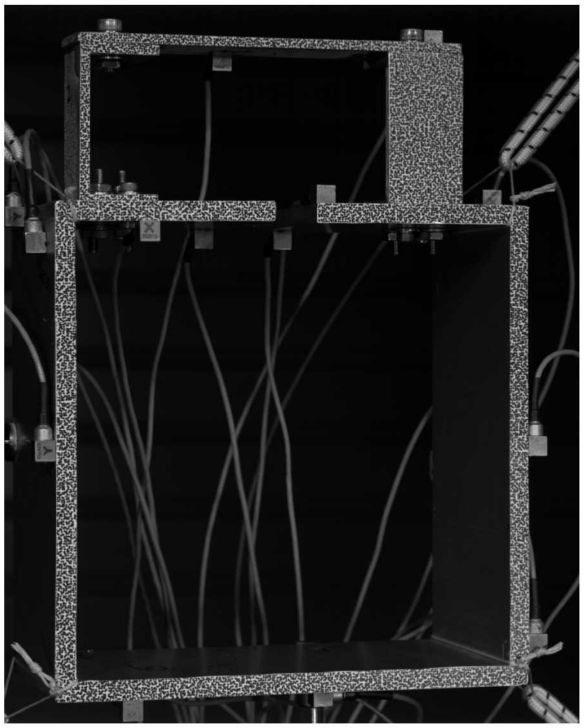


Figure 23: Unwrapping the 3D mesh by projecting to image coordinates (left) from the 3D view (right).

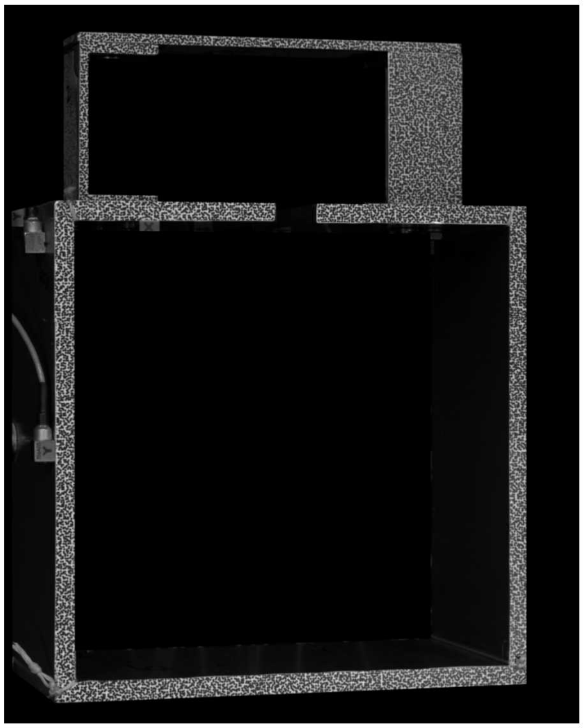
While the speckle pattern imposed on the BARC was derived from the optical test simulator, it is likely that the speckle pattern was not exactly reproduced; the sticker could be placed slightly off-center or could have small bends in it. However, in some cases, it may be advantageous to have the optical test simulator exactly reproduce the contrast pattern from the image. The material texture was switched from the image of the speckle pattern to the reference image from the test and it is loaded into the UV/Image Editor in Blender. The view was set to the left camera, and the 3D model was unwrapped by projecting the coordinates from the current view, this is shown in Figure 23. Note that this projection could also be performed algorithmically using the camera equation (4). This procedure was performed for each object in the Blender scene. Note that the reference image from the test will almost certainly have lens distortions baked in, and depending on how severe they are, the reference image may not project well to the mesh. In this example, the lens distortions were removed from the reference image using the `undistort` function in OpenCV [14] prior to projecting the finite element coordinates to the image.

The last step to reproduce an optical test in the optical test simulator is to adjust the lighting. Ideally, identical lighting to the object under test is desired. This lighting is already baked into the image texture that was just applied to the finite element geometry, so only a flat-white, uniform light is required. This can be achieved using environmental lighting in Blender. If using the Internal render engine, the render background can be specified independently of the environmental lighting. If using the Cycles render engine, setting the environmental lighting to white will also change the background to white, which is likely not what was obtained in the test. This can be fixed by placing a large diffuse black plane in the scene behind the part. Figure 24 shows the comparison between the true test image and the optical test simulator image.

Arbitrary displacements can then be applied to the optical test simulator mesh to develop processing parameters for DIC or other optical techniques on "perfect" images where the analytical deformation is known in advance. Figure 25 shows two mode shapes of the BARC finite element model rendered at 50 pixels displacement using the optical test simulator. Note that the image has projected the accelerometers placed on the test article onto the finite element mesh, but there are no accelerometers in the mesh. When large displacements occur these poorly projected portions of the image will become distorted. However, in this application, the only surfaces of interest were the speckled surfaces, so there was no issue with the distortions in the accelerometer renderings. If multiple renders of an object are required from multiple angles, it may be necessary to texture different surfaces of the model separately using reference images from those camera views.

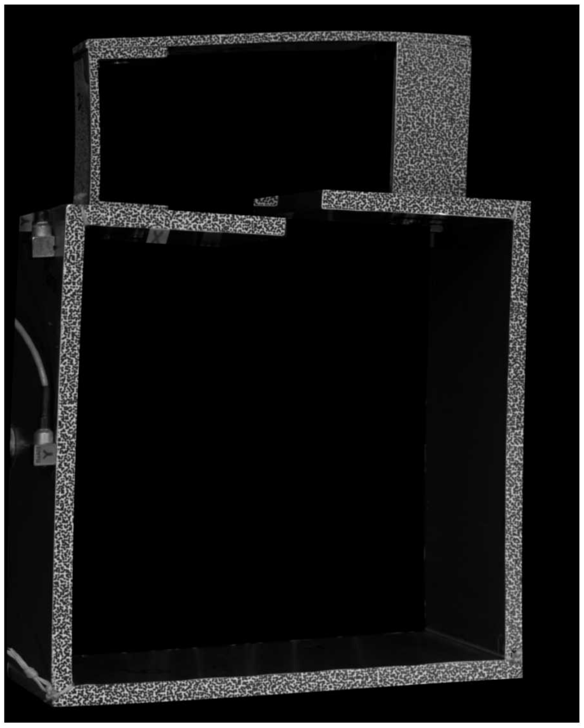


(a) Actual camera image of BARC

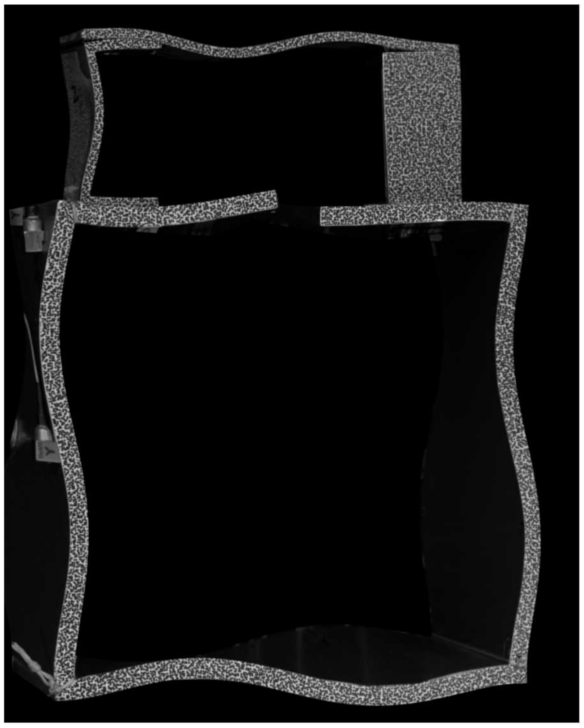


(b) Synthetic optical test simulator image of BARC

Figure 24: Comparison of actual camera image and synthetic image from the simulator.



(a) Mode 7



(b) Mode 36

Figure 25: Two grossly exaggerated mode shapes rendered by the optical test simulator. Note that due to the projective application of the image texture from the camera view, the textures on faces perpendicular to the image plane of the camera are distorted. If large displacements are expected using the optical test simulator, these faces could be textured separately using images from different camera angles.

7 Conclusions and Future Work

This work demonstrated a tool set to create synthetic images for evaluation of optical techniques using a mature, open-source framework with a graphical user interface as well as a powerful programming interface that enables tedious tasks such as deforming meshes, moving calibration targets, and rendering images to be automated. A simple plate workflow and DIC analysis was presented which demonstrated the capability of the Blender software to generate accurate subpixel images that can be utilized in DIC analyses. The subpixel resolution analysis demonstrated that the simulator would not add in significant error sources if proper rendering parameters were selected. The test planning example demonstrated the ability of the simulator to match experimental parameters such as camera parameters and utilize experimental images as the texture applied to the mesh in the simulator. This allows the simulator to analyze an experimental setup, perhaps to determine what image resolution might be necessary to resolve a specific displacement or to examine if a given lens focal length will result in the desired field of view. Additionally, it has been used to directly create speckle patterns that can be applied to test articles; the speckle size can easily be modified by scaling UV coordinates of the mesh on the image to achieve proper pixel-per-speckle ratios, and the speckle pattern used in the virtual experiment can be printed to sticker paper and adhered directly to the test article.

This paper has barely scratched the surface of the photo-realistic image effects that can be implemented using Blender or added via post-processing. Future work will focus on investigating the issues that often arise in a DIC test. Limited depth of field, specular reflections, and light refraction due to heat waves from bright lights are some examples of non-ideal conditions that can be implemented inside the Blender software. Additional non-ideal conditions such as lens distortions and image noise can easily be added by post-processing the image.

References

[1] M. Potmesil and I. Chakravarty, "Synthetic image generation with a lens and aperture camera model," *ACM Transactions on Graphics (TOG)*, vol. 1, no. 2, pp. 85–108, 1982.

[2] J.-J. Orteu, D. Garcia, L. Robert, and F. Bugarin, "A speckle texture image generator," in *Speckle06: speckles, from grains to flowers*, vol. 6341, p. 63410H, International Society for Optics and Photonics, 2006.

[3] D. Garcia, J.-J. Orteu, L. Robert, B. Wattrisse, and F. Bugarin, "A generic synthetic image generator package for the evaluation of 3D Digital Image Correlation and other computer vision-based measurement techniques," in *PhotoMechanics 2013*, (Inconnue, France), p. Clé USB, May 2013.

[4] P. Lava, S. Cooreman, S. Coppieters, M. D. Strycker, and D. Debruyne, "Assessment of measuring errors in dic using deformation fields generated by plastic fea," *Optics and Lasers in Engineering*, vol. 47, no. 7, pp. 747 – 753, 2009.

[5] M. Rossi, M. Badaloni, P. Lava, D. Debruyne, G. Chiappini, and M. Sasso, "Advanced test simulator to reproduce experiments at small and large deformations," in *Advancement of Optical Methods in Experimental Mechanics, Volume 3* (H. Jin, C. Sciammarella, S. Yoshida, and L. Lamberti, eds.), (Cham), pp. 27–33, Springer International Publishing, 2014.

[6] R. Balcaen, L. Wittevrongel, P. L. Reu, P. Lava, and D. Debruyne, "Stereo-DIC calibration and speckle image generator based on fe formulations," *Experimental Mechanics*, vol. 57, pp. 703–718, Jun 2017.

[7] The Blender Foundation, "Blender.org." `https://www.blender.org/`. Accessed 2019-02-24.

[8] G. Varol, J. Romero, X. Martin, N. Mahmood, M. J. Black, I. Laptev, and C. Schmid, "Learning from synthetic humans," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4627–4635, July 2017.

[9] The Pixelary, "Blender for computer vision machine learning." `http://blog.thepixelary.com/post/174286685782/blender-for-computer-vision-machine-learning`, May 2018. Accessed: 2019-03-18.

[10] International Digital Image Correlation Society, *A Good Practices Guide for Digital Image Correlation*. Oct. 2018.

[11] D. Z. Turner, "Digital image correlation engine (DICe) reference manual," Sandia Report SAND2015-10606 O, Sandia National Laboratories, 2015.

[12] H. W. Schreier, J. R. Braasch, and M. A. Sutton, "Systematic errors in digital image correlation caused by intensity interpolation," *Optical Engineering*, vol. 39, no. 11, pp. 2915 – 2921 – 7, 2000.

[13] D. E. Soine, R. J. Jones, Jr., J. M. Harvie, T. J. Skousen, and T. F. Schoenherr, "Designing hardware for the boundary condition round robin challenge," in *Topics in Modal Analysis & Testing* (M. Mains and B. Dilworth, eds.), vol. 9 of *Conference Proceedings of the Society for Experimental Mechanics Series*, (Cham), pp. 119–126, Springer, 2019.

[14] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[15] K. S. Arun, T. S. Huang, and S. D. Blostein, "Least-squares fitting of two 3-d point sets," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-9, pp. 698–700, sep 1987.

[16] O. Sorkine-Hornung and M. Rabinovich, "Least-squares rigid motion using SVD," *Computing*, vol. 1, no. 1, 2017.