Technical Report Documentation Page

1. Report No. ABC-UTC-2013-C3-FIU02	2. Government Accession No.	3. Recipient's Catalog No.	
4. Title and Subtitle **Predictive Computer Program for Proactive Demolition Planning**		5. Report Date May 31, 2019 6. Performing Organization Code	
7. Author(s) Ali Bakhtiari (orcid.org/0000-0003-4286-5400) Seung Jae Lee (orcid.org/0000-0002-2180-3502)		8. Performing Organization Report No. ABC-UTC-2013-C3-FIU02	
9. Performing Organization Name and Address Department of Civil and Environmental Engineering Florida International University 10555 West Flagler Street, EC 3680 Miami, FL 33174		10. Work Unit No. (TRAIS) 11. Contract or Grant No. DTRT13-G-UTC41	
12. Sponsoring Organization Name and Address Accelerated Bridge Construction University Transportation Center Florida International University 10555 W. Flagler Street, EC 3680 Miami, FL 33174	US Department of Transportation Office of the Assistant Secretary for Research and Technology And Federal Highway Administration 1200 New Jersey Avenue, SE Washington, DC 201590	13. Type of Report and Period Covered Final (06/01/17 – 05/31/19) 14. Sponsoring Agency Code	
15. Supplementary Notes Visit www.abc-utc.fiu.edu for other ABC reports.			
16. Abstract Bridges represent a significant subpopulation of our civil infrastructure. Majority of them are deteriorating fast and in need of replacement or rehabilitation. The first step of those replacement/rehabilitation projects is typically to either entirely or partly demolish the existing structure. Therefore, proactive planning for controlled demolition is of utmost importance to proceed with the rest of construction project in a timely manner. Maintaining the integrity of neighboring infrastructure, e.g., permanent roadways, nearby transmission lines, and the safety of workers are critical issues, for which contingency plans also must be developed based on any feasible emergency scenarios. It is typically hard to develop a general guideline/specification that can facilitate safe and efficient demolition, and very limited information has been available to guide structural engineers and contractors on how to proceed with the demolition of an existing structure. This lack of generalized procedure has led to structural engineers and contractors approaching the demolition work differently, and as a result, most states neither specify parameters for demolition equipment nor require the submission of contractor qualifications with the demolition plans. The potential hazards and inefficiency may be better controlled and possibly eliminated by leveraging computer simulation that can help realistically predict the demolition process. However, such computer tools for simulation-aided demolition remain to be developed in the bridge engineering community. This study aims to enhance the predictive capabilities by exploring the numerical simulation techniques that can realistically model, simulate and visualize the bridge demolition, which will better support the engineers and contractors' decision making.			
17. Key Words Bridge demolition planning; Numerical modeling;		18. Distribution Statement No restrictions.	
19. Security Classification (of this report) Unclassified.	20. Security Classification (of this page) Unclassified.	21. No. of Pages 104	22. Price

Form DOT F 1700.7 (8-72) Reproduction of completed page authorized

(this page is intentionally left blank)

Predictive Computer Program for Proactive Demolition Planning

Final Report

May 2019

Principal Investigator: Seung Jae Lee

Department of Civil and Environmental Engineering

Florida International University

Authors

Ali Bakhtiari

Seung Jae Lee

Sponsored by

Accelerated Bridge Construction University Transportation Center

A report from

Department of Civil and Environmental Engineering

Florida International University

Miami, FL

DISCLAIMER

The contents of this report reflect the views of the authors, who are responsible for the facts and the accuracy of the information presented herein. This document is disseminated in the interest of information exchange. The report is funded, partially or entirely, by a grant from the U.S. Department of Transportation's University Transportation Program. However, the U.S. Government assumes no liability for the contents or use thereof. In addition, the authors make no warranty, representation, or guarantee, either express, implied, statutory or otherwise, with respect to the software developed and discussed in this project.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This project was supported by the Accelerated Bridge Construction University Transportation Center (ABC-UTC at www.abc-utc.fiu.edu) at Florida International University (FIU), as lead institution, and Iowa State University (ISU) and the University of Nevada-Reno (UNR) as partner institutions. The authors would like to acknowledge the ABC-UTC support. The authors would like to extend special appreciation to the ABC-UTC and the U.S. Department of Transportation Office of the Assistant Secretary for Research and Technology for funding this project. The authors would like to thank all the State DOTs that participated in the survey; this work would not have been possible without their participation. The authors would like to thank the Research Advisory Panel members.

Chapter 1. INTRODUCTION

1.1. Project Motivation

Bridges represent a significant subpopulation of our civil infrastructure. Majority of them are deteriorating fast and in need of replacement or rehabilitation. The first step of those replacement/rehabilitation projects is typically to either entirely or partly demolish the existing structure. Therefore, proactive planning for controlled demolition is of utmost importance to proceed with the rest of construction project in a timely manner. Maintaining the integrity of neighboring infrastructure (e.g., permanent roadways, nearby transmission lines) and the safety of workers are critical issues, for which contingency plans also must be developed based on any feasible emergency scenarios.

However, little effort has been given to develop better removal techniques of existing structures, while great effort has been made to design/construction techniques for new structures. Planning failure is often unpredictably realized in the demolition project due to inherent uncertainty hard to characterize ahead, not only in the deteriorated condition of the structure that may be far different from that of the original design, but also in the mode of destruction that may depend on adopted demolition methods, types and performance of destruction tools, dismantling sequence and associated change in the remaining structural capacity during the demolition process.

It is typically hard to develop a general guideline/specification that can facilitate safe and efficient demolition, and very limited information has been available to guide structural engineers and contractors on how to proceed with the demolition of an existing structure. This lack of generalized procedure has led to structural engineers and contractors approaching the demolition work differently, and as a result, most states neither specify parameters for demolition equipment nor require the submission of contractor qualifications with the demolition plans [1]. The potential hazards and inefficiency may be better controlled and possibly eliminated by leveraging computer simulation that can help realistically predict the demolition process. However, such computer tools for simulation-aided demolition remain to be developed in the bridge engineering community.

1.2. Research, Objectives, and Tasks

The primary objective of this research project is to enhance the predictive capabilities by exploring a set of numerical simulation techniques which have led to development of a computer program add-on named as Bridge Demolition Add-on (v1.0.0) that enables to realistically model, simulate

and visualize the bridge demolition, which will better support the engineers and contractors' decision making. This add-on has been developed as an add-on to Blender, a free and open-source 3D computer graphics software that uses the impulse-based Bullet engine, a physics library which is also free and open-source. These objectives are tackled through the following research tasks.

- *Task 1 – Literature review*: This task aims to make comprehensive literature review regarding the relevant research efforts that have been made to numerically simulate bridge demolition process, for which related literature has been extensively reviewed.
- *Task 2 – Design of simulation components*: This task aims to design the required simulation components to reproduce various demolition scenarios. In particular, balancing between simulation fidelity and computational efficiency will be pursued as the simulation targets solving the field scale problems for use in the engineering practice, which is different from a conventional fracture mechanics solver that focuses on the precise crack propagation at a relatively small scale.
- *Task 3 – Code development*: This task aims to implement the numerical components necessary for the simulations. The implementation will be done as add-on in an object-oriented manner for the code extensibility to facilitate the implementation of any further ideas even after this project is completed.
- *Task 4 – Verification and simulation*: This task verifies the numerical issues in the developed program. Any major programming issues and errors will be resolved and debugged through the verification process. The program will be then validated using available data and video recordings obtained from the previous demolition projects.
- *Task 5 – Final Report*: A final report will be prepared meeting the RITA requirements for UTC funded projects. The content of the report will contain a detailed summary of the results from the preceding tasks and a recommendation for future phases of the project, if necessary.

1.3. Research Advisory Panel (RAP)

The project work and the developed survey were done with support of the Research Advisory Panel (RAP). The following people are on the RAP:

- Benjamin Beerman (FHWA) and James Corney (Utah DOT)

1.4. Report Overview

This project is intended to explore the numerical simulation techniques for bridge demolition which have led to the development of a computer program add-on named as Bridge Demolition Add-on (v1.0.0) hereafter simply BDA written in Python using Blender's API. The codes in this project builds upon Blender, a free and open-source 3D computer graphics software toolset under GNU General Public License, and Bullet, a free and open-source physics engine library under zlib license. This project runs, modifies, and shares the codes under these licenses.

This report aims to help engineers can leverage the software package to proactively plan the bridge demolition project. The simulation is demonstrated for the demolition using impact loading (e.g., wrecking ball) and explosive loading. This report will discuss the key algorithms developed for such demolition simulations and the modeling procedure leveraging BDA along with Blender and Bullet for the demolition modeling and visualization of various scenarios. The proposed approach using the computer simulation will help optimize the demolition planning using the explosives with the stress contours obtained from the simulations. The explosion is simulated based on the theory of blast waves, and a customized MATLAB code provides analysis for the optimization.

The report provides how to use the software package with examples to enhance the predictive capabilities. Hence, it makes it easier for decision-makers to get better prepared for the worst-case scenarios and choose the best demolition planning ahead depending on the target structure's configuration, e.g., number of spans, span length, weight of structure as well as the environmental considerations, time limits, budget, by proactively performing the visualized demolition scenarios.

1.5. Research Approach and Methods

Simulation of bridge demolition is a challenging problem that requires to numerically model initiation and propagation of cracks in the material due to interaction between the bridge and the demolition equipment or the explosive blast waves, and break-up of the structure into pieces. Corresponding deformation of damaged bridge and displacement of debris are physically complex phenomena, for which material transition from a continuous medium to multiple broken (discontinuous) pieces needs to be realistically modeled. To this end, this project adopts a discrete mechanical simulation technique for high-fidelity simulation of such destruction problems, which commonly model a continuum as a discrete system that is composed of bonded rigid or deformable elements.

Chapter 2. DESCRIPTION OF RESEARCH PROJECT TASKS

2.1. Literature Review

One of the objectives of this task is to investigate any available literature published for demolition planning. However, there are few guidelines or specifications in hand and this is exactly the one of the very motivations of this study to contribute to a comprehensive demolition guideline for civil engineers and contractors in the field. Among the previous efforts, published as an ABC-UTC report, a nationwide survey was conducted for gathering data and assessing current demolition practices in the states. According to results of the survey, less than 40 percent of the DOTs responded to survey have guidelines or specifications for demolition planning, less than half of them do not have pre-demolition decision making sessions and more than 75 percent do not investigate for any demolition equipment parameters. Therefore, about 40 percent of the states have suffered from accidental collapse in the last 15 years. This indicates that demolition issues are common among states and also intensifies the necessity of having guidelines for demolition planning since these accidental effects can result in fatalities, loss of life and other issues such as traffic interruption and harming surrounding infrastructure [2].

There are few guidelines available in the field for demolition planning for bridges, but they are mostly restricted to a specific case. In a guideline published for demolition project of Burlington Northern Santa Fe railroad, general considerations for contractors during demolition are listed and the sequential tasks to bed done for bridge removal, tips for managing debris, and consideration regarding cranes and utilities are briefly provided [3]. However, these case studies are not providing comprehensive demolition planning guideline which can be used widely among all of the civil engineers and contractor of bridge demolition projects.

The other objective of this task is to make comprehensive literature review regarding the relevant research efforts that have been made to simulate bridge demolition process using discrete mechanical simulation technique. Related literature has been extensively reviewed, and summary of the findings is provided as below:

Based on the literature review, it was concluded that there are largely two different types of approaches can be made depending on the order of dynamics adopted for the simulation: (i) force-based or (ii) impulse-based dynamic approach.

(i) Force-based dynamic approach has been widely adopted for discrete mechanical simulations. The interaction of modeled elements is explicitly considered through a set of springs and dashpot, thus often called mass-spring-dashpot system. The bond is modeled to be broken if the tensile force exceeds a threshold, and elements separated, which is the major difference from continuum mechanics frameworks such as the finite element method (FEM). The discrete element method (DEM) is currently the most popular force-based numerical method in the field of computational discrete mechanics, and has been widely adopted in a number of cross-disciplinary applications to model the discontinuities in material and structural systems, e.g., collapse of unreinforced masonry structures [4], material fracture [5], [6], progressive collapse of building structures [7], etc. The force-based (or acceleration-based) dynamics approach is based on the ordinary 2nd order dynamics (i.e., $f = ma$), thus computationally demanding in general especially with the required simulation fidelity. The time step size Δt is also limited by the stability criterion for explicit numerical time integration which is, in turn, determined by modeled element size. In this way, a very small Δt is typically used in DEM simulation coupled with needed double-precision for the numerical stability can result in significant computational costs. This major computational bottleneck has significantly limited DEM application to relatively small-scale engineering problems.

(ii) Impulse-based dynamic approach is the other class of discrete mechanical simulation technique often adopted to realistically present larger scale discrete bodies' interactions. The impulse-based dynamics employs a reduced 1st order dynamics to directly manipulate the velocity of discrete elements (i.e., $i = m\Delta v$) compared to the 2nd order dynamics that works on the integration of accelerations as used in the conventional DEM. The time step size Δt is limited by the physics of the problem whereby a too large time step size will result in elements passing through each other, but not limited by numerical stability and very small Δt unlike DEM. Hence, significant numerical efficiency is demonstrated with almost two orders of speed-up, while the physical plausibility is still maintained. Consequently, it takes only a few days to simulate a large-scale multi-body problem, which typically takes several months if the computationally intensive DEM is used to simulate the same problem. However, the contact force is not an integral part of simulation, which is the major drawback of the impulse-based dynamics approach, as the primary variables are collision impulse and velocity, not contact force and acceleration that are still required for engineering applications. Recently, impulse-based DEM (iDEM) was developed based on the impulse-based dynamics that retrieves the 2nd order engineering details lost (i.e., contact force) due to the order

reduction in the equation of motion, which enabled the impulse-based dynamics suitably adopted for engineering simulations [8]. As an incentive, the impulse-based dynamics approach will allow for more simulations to be performed for a given time while physical plausibility being maintained. Therefore, this approach will enable to better characterize and identify the most likely critical scenarios in a demolition project ahead.

2.2. Design of Simulation Components

The objective of this task is to design overall architecture of the program, which includes various numerical components required to simulate the demolition. In particular, balancing between simulation fidelity and computational efficiency will be pursued as the computational framework to be developed in this study targets solving a field scale problem for use in the engineering practice, which is different from a conventional fracture mechanics solver that focuses on the precise crack propagation at a relatively small scale.

Figure 1 overviews the overall calculation flow of the simulation framework, which is mainly composed of 2 parts, i.e., the impulse-based dynamic simulation (Part 1) and the retrieval of the 2nd order engineering details (Part 2). The 1st part corresponds to Stages 1 to 6, while the 2nd part corresponds to Stage 7 in the figure. This framework adopts the conventional algorithm/code used in open source physics engines, e.g., Bullet [9], for the impulse-based dynamic simulation (Part 1), and adopts/modifies the algorithm in [8] to develops the retrieval part as an add-on (Part 2).

In Stage 1, the initial conditions such as element shapes, orientations, sizes and other modeling properties are defined such as the coefficient of restitution. Voronoi tessellation [10] is adopted for the domain discretization and the pre-fragmentation of bridge members to be destructed. The initial list of neighboring elements will be then developed. In Stage 2, the initial velocity is updated by integrating the body force over the given Δt. This initial velocity calculation will be mostly used for the trajectory motion update of the demolition debris after onset of breaking and the demolition tools such as wrecking ball. In Stage 3, the code performs the neighbor search to find nearest elements. The list of close pairs will be updated as the simulation of demolition proceeds. A general neighbor search algorithm such as Two Level Search [11] will be used. In Stage 4, more accurate geometric test is performed to find the contact points between colliding element pairs. In Stage 5, the collision impulse is computed, for which the solver iteratively goes through all the contact points between the colliding element pairs until the specified collision law is numerically

satisfied. The velocity from collision is then updated. In Stage 6, the element position is finally updated. In Stage 7, i.e., Part 2, the 2nd order engineering details are retrieved from the computed collision impulse.

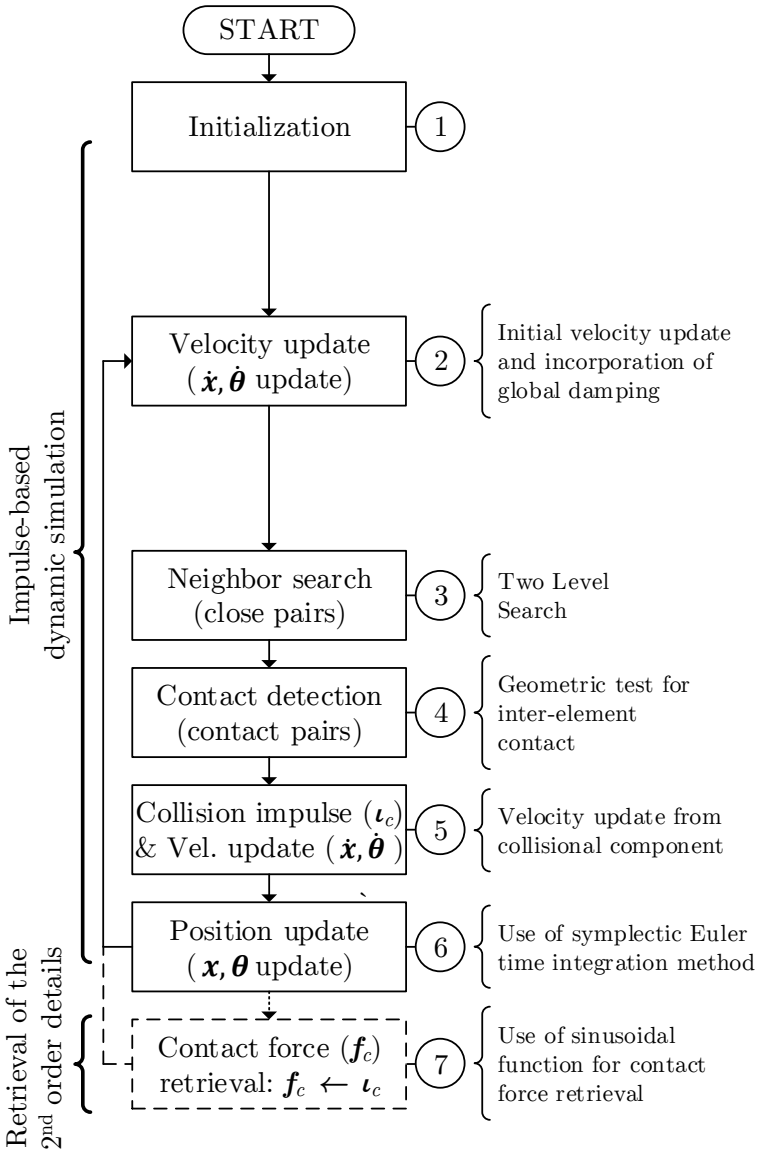

Figure 1. Calculation flow of the simulation framework with the stage number shown in circle

2.3. Code Development

2.3.1. Bridge Demolition Add-on Overview

Bridge Demolition Add-on (BDA) version 1.0.0 has been developed from this project using Blender's Python API [12]–[14]. It was designed as an add-on to provide enhancement to Blender 2.79, whereby the computational framework is customized for numerical modeling and simulation of various bridge demolition scenarios.

BDA helps model two popular demolition methods for bridge demolition: (i) explosive demolition and (ii) mechanical demolition with a wrecking ball. BDA is also equipped with additional post-processing features such as contact force computation in the structure during demolition. The Blender's physics engine Bullet works on the first order dynamics where the collision impulse is the primary variable, where the contact force computation is bypassed during the simulation. However, the force is the key engineering information for demolition planning, e.g., to determine the weight of wrecking ball, size of equipment, etc., and therefore is of great importance from the structural engineering aspects. The developed add-on enables to retrieve the contact force, i.e., the second order details, and this information can be visualized using the contour with BDA. Moreover, BDA is customized to help simulate the demolition by explosives. BDA provides the user interface dialog box that enables user to enter modeling input parameters for explosion demolition simulation, and the post-processing panel for contact force, predicted explosion loading, estimated debris generation and force/stress contour visualization as the outputs. In the demolition using explosives, a major concern might be related to using the minimum possible explosive energy that is enough to destroy the structure given the budget for the enhanced efficiency and to minimally impact the environment with less debris missiles, which will in turn protect the neighboring infrastructure as well as workers' safety from the demolition induced hazards. BDA will help address these issues. Another significant feature is Debris Propagation Log, which enables the decision-makers, engineers and contractors to know how far the demolition induced debris travels. BDA user interface and features are further discussed in following sections.

The implementation of BDA is done in an object-oriented manner for the code extensibility to facilitate implementation of any further ideas even after this project is completed. This project does not consider hardware acceleration technique such as general-purpose computing on graphics processing units due to the limited project timeframe. However, the implementation will be made

with possibility open to the parallel computation capability that may be added later. The code development is made upon Blender [15], which is under GNU General Public License. Blender uses Bullet, which is under zlib license, as internal physics engine. Therefore, this project runs, modifies, and shares the derivatives under these licenses.

Modeling the geometry of the structure, assigning mechanical modeling properties, discretization based on Voronoi algorithm and creating constraints among fractured rigid bodies are performed in Blender, the developed Bridge Demolition Add-on (BDA) that works with Blender makes easier for decision-makers to get better prepared for the worst-case scenarios and choose the best demolition planning ahead depending on the target structure's configuration, e.g., number of spans, span length, weight of structure as well as the environmental considerations, time limits, budget, by proactively performing the visualized demolition scenarios.

2.3.2. BDA's User Interface (UI) Overview, Features and Code Development

BDA can be installed and added to Blender as shown in Figure 2 to 5. Open Blender, go to the File tab, select User Preferences, and select Install Add-on from File in the Add-ons section (Figure 2). Then select the path to the zip file of BDA downloaded on your computer (Figure 3). Select the add-on file and install it. Then the add-on name and details are shown (Figure 4). Save User Settings, and close Blender User Preferences Window. Then Bridge Demolition Add-on is shown on the Blender's tool bar as BDA (Figure 5).

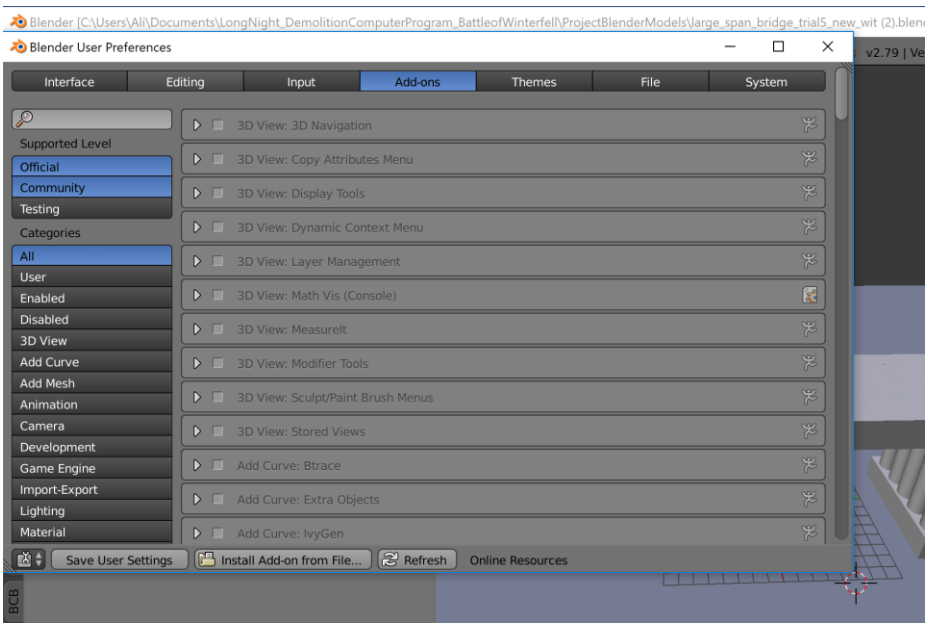

Figure 2. Blender user preferences, Add-ons

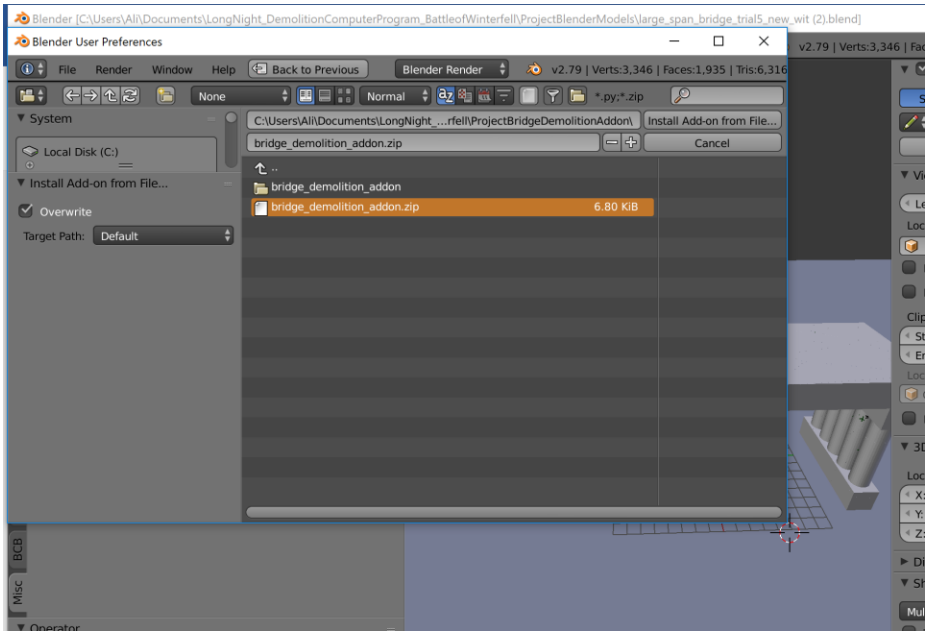

Figure 3. Installing Bridge Demolition Add-on (v 1.0.0)

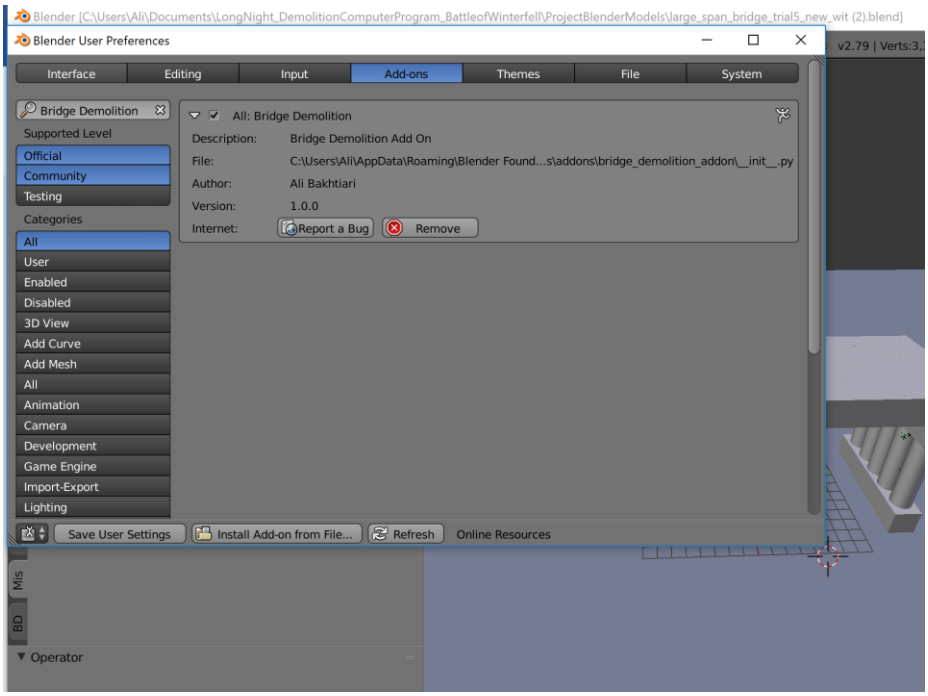

Figure 4. Bridge Demolition Add-on's name and description are shown for installation

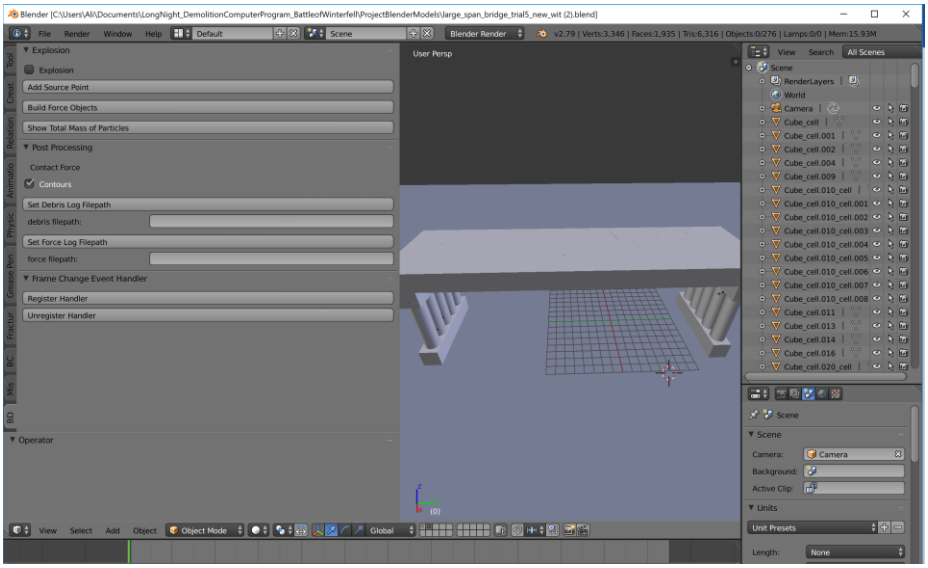

Figure 5. BDA added to Blender's UI

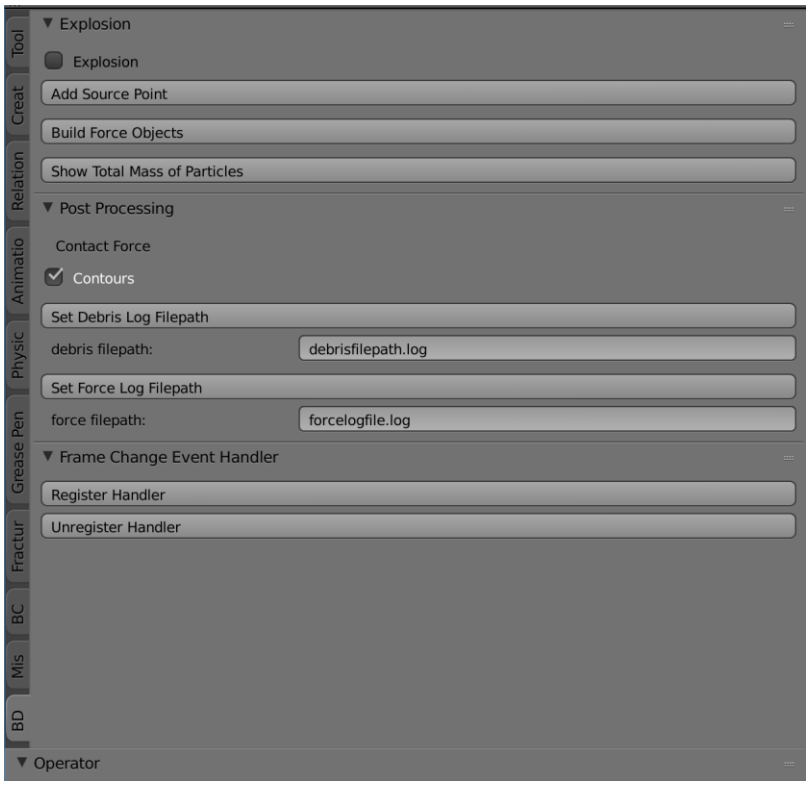

Figure 6. BDA UI

Figure 6 shows the graphical user interface created for Bridge Demolition Add-on. As shown in the figure, the BDA UI has three major panels: *Explosion*, *Post Processing* and *Frame Change Event Handler*.

In Explosion panel, there is a check box named Explosion. If the user is going to perform a simulation of demolition by explosives, this box should be checked. It is worth to note accidentally checking this box causes no problem even if the demolition method is not by explosion. Instead, it will just slow down the computation process because of some initializations required for explosion simulation. User can control the simulation of explosives with two parameters. The first parameter is the location of explosives (or explosion source points). With Add Source Point clicked in the UI, the user can define unlimited numbers of explosion source points. Figure 6 shows a new row is generated with the Add Source Point activated, and the user can define explosion source point location in Cartesian coordinate (with X, Y and Z coordinates) and the amounts of explosives related to the entered source points. The amount is to be entered as the TNT mass in kg.

After defining desired explosion source points, Build Force Objects should be clicked to create a set of empty objects to be used for applying explosive force in 3D. This will be further elaborated in following sections. User can click Show Mass of Particles to calculate the total mass of bridge model, which is optional.

Post Processing panel introduces two additional features. The panel is mainly to compute the contact forces among fractured bodies. Contours check box enables the contact force visualization via contours during the simulation. The Set Force Log Filepath option is to write the computed contact forces in each time step of simulation on a file whose path is selected by the user. The Set Debris Log Filepath option will record the locations of rigid bodies in the first and last time steps on a file whose path is selected by the user. This will allow to calculate maximum distance traveled by a fractured part and enable to predict debris impact on the surroundings due to the demolition. This information will be vital in the demolition project because stakeholders may want to make sure the surrounding infrastructure is not affected by demolition-induced debris.

The Frame Change Event Handler panel is designated for registering and unregistering event frame handlers. Event frame handlers play important role to transfer data and calculated values from one time step to another to be used for the contour development. BDA User Interface (UI) is coded in Python using Blender's API, which is shown on the next pages:

```
1.  ##############################################################################
2.  # Panel UI
3.  ##############################################################################
4.  bl_info = {
5.      'name': 'Saved Variables',
6.      'category': 'All'
7.  }
8.
9.  import bpy
10. class CustomDrawOperator(bpy.types.Operator):
11.     bl_idname = "object.custom_draw"
12.     bl_label = "Import"
13.
14.     filepath = bpy.props.StringProperty(subtype="FILE_PATH")
15.
16.     my_float = bpy.props.FloatProperty(name="Float")
17.     my_bool = bpy.props.BoolProperty(name="Toggle Option")
18.     my_string = bpy.props.StringProperty(name="String Value")
19.
20.     def execute(self, context):
21.         print("filepath:", self.filepath)
22.         bpy.context.scene.explosion_tool.debrisfilepath = self.filepath
23.         return {'FINISHED'}
24.
25.     def invoke(self, context, event):
26.         context.window_manager.fileselect_add(self)
27.         return {'RUNNING_MODAL'}
28.
29.     def draw(self, context):
30.         layout = self.layout
31.         col = layout.column()
32.         col.label(text="Custom Interface!")
33.
34.         row = col.row()
35.         row.prop(self, "my_float")
36.         row.prop(self, "my_bool")
37.
38.         col.prop(self, "my_string")
39.
40.
41. class CustomDrawOperator2(bpy.types.Operator):
42.     bl_idname = "object.custom_draw2"
43.     bl_label = "Import"
44.
45.     filepath = bpy.props.StringProperty(subtype="FILE_PATH")
46.
47.     my_float = bpy.props.FloatProperty(name="Float")
48.     my_bool = bpy.props.BoolProperty(name="Toggle Option")
49.     my_string = bpy.props.StringProperty(name="String Value")
50.
51.     def execute(self, context):
52.         print("filepath:", self.filepath)
53.         bpy.context.scene.explosion_tool.logfilepath = self.filepath
54.         return {'FINISHED'}
55.
56.     def invoke(self, context, event):
57.         context.window_manager.fileselect_add(self)
58.         return {'RUNNING_MODAL'}
59.
60.     def draw(self, context):
61.         layout = self.layout
```

```
        col = layout.column()
        col.label(text="Custom Interface!")

        row = col.row()
        row.prop(self, "my_float")
        row.prop(self, "my_bool")

        col.prop(self, "my_string")


class MessageBox(bpy.types.Operator):
    bl_idname = "message.messagebox"
    bl_label = ""

    message = bpy.props.StringProperty(
        name = "message",
        description = "message",
        default = ''
    )

    def execute(self, context):
        self.report({'INFO'}, self.message)
        print(self.message)
        return {'FINISHED'}

    def invoke(self, context, event):
        return context.window_manager.invoke_props_dialog(self, width = 400)

    def draw(self, context):
        self.layout.label(self.message)
        self.layout.label("")



class SceneItems(bpy.types.PropertyGroup):
    source = bpy.props.FloatVectorProperty()

       class AmountItems(bpy.types.PropertyGroup):
           amount = bpy.props.FloatProperty()

       class DebrisLogFileOperator(bpy.types.Operator):
           bl_idname = "scene.debris_log_file_operator"
           bl_label = "Set Debris Log Filepath"

           def execute(self, context):
               bpy.ops.object.custom_draw('INVOKE_DEFAULT')
               return {'FINISHED'}

       class ForceLogFileOperator(bpy.types.Operator):
           bl_idname = "scene.logfilepath_operator"
           bl_label = "Set Force Log Filepath"

           def execute(self, context):
               bpy.ops.object.custom_draw2('INVOKE_DEFAULT')
               return {'FINISHED'}


       class MassButtonOperator(bpy.types.Operator):
           bl_idname = "scene.mass_button_operator"
           bl_label = "Show Total Mass of Particles"
```

```
123.
124.             def execute(self, context):
125.                 msg = 'Total Mass: ' + str(total_mass())
126.                 bpy.ops.message.messagebox('INVOKE_DEFAULT', message = msg)
127.                 return {'FINISHED'}
128.
129.
130.         class AddButtonOperator(bpy.types.Operator):
131.             bl_idname = "scene.add_button_operator"
132.             bl_label = "Add Source Point"
133.
134.             def execute(self, context):
135.                 id = len(context.scene.collection)
136.                 new = context.scene.collection.add()
137.                 new.name = str(id)
138.                 new.source = (0,0,0)
139.                 newa = context.scene.amountcollection.add()
140.                 newa.amount = 0
141.                 return {'FINISHED'}
142.
143.         class BuildButtonOperator(bpy.types.Operator):
144.             bl_idname = "scene.build_button_operator"
145.             bl_label = "Build Force Objects"
146.
147.             def execute(self, context):
148.
149.                 build_all_force_objects()
150.                 return {'FINISHED'}
151.
152.
153.
154.         class ButtonOperator(bpy.types.Operator):
155.             bl_idname = "scene.button_operator"
156.             bl_label = "Button"
157.
158.             id = bpy.props.StringProperty()
159.
160.             def execute(self, context):
161.                 print("Pressed button ", self.id)
162.                 for item in context.scene.collection:
163.                     print(item.value[0])
164.                 return {'FINISHED'}
165.
166.         class DeleteOperator(bpy.types.Operator):
167.             bl_idname = "scene.delete_operator"
168.             bl_label = "Delete"
169.
170.             ids = bpy.props.IntProperty()
171.
172.             def execute(self, context):
173.                 context.scene.collection.remove(self.ids)
174.                 context.scene.amountcollection.remove(self.ids)
175.                 return {'FINISHED'}
176.
177.         class FancyPanel(bpy.types.Panel):
178.             bl_idname = "panel.panel_fancy_panel"
179.             bl_label = "Explosion"
180.             bl_space_type = "VIEW_3D"
181.             bl_region_type = "TOOLS"
182.             bl_category = "BDA"
183.
```

```
184.
185.             def draw(self, context):
186.                 layout = self.layout
187.                 row = layout.row()
188.                 row.prop(bpy.context.scene.explosion_tool, "explosion_flag")
189.                 self.layout.operator("scene.add_button_operator")
190.                 row = layout.row()
191.                 i = 0
192.                 for item in context.scene.collection:
193.                    row = self.layout.row(align=True)
194.                    row.prop(item, "source", text="source " + str(i))
195.                    row.prop(context.scene.amountcollection[i], "amount")
196.                    row.operator("scene.delete_operator", text="Delete").ids = i
197.                    i = i + 1
198.
199.                 row = layout.row(align=True)
200.                 row.operator("scene.build_button_operator")
201.                 row = layout.row()
202.                 self.layout.operator("scene.mass_button_operator")
203.
204.
205.         # ------------------------------------------------------------------------
206.         #    Scene Properties
207.         # ------------------------------------------------------------------------
208.
209.         class ExplosionVariables(bpy.types.PropertyGroup):
210.             logfilepath = bpy.props.StringProperty(
211.                 name = "force filepath",
212.                 description = "write to file total contact force values"
213.             )
214.             debrisfilepath = bpy.props.StringProperty(
215.                 name = "debris filepath",
216.                 description = "write to file debris propagation data"
217.             )
218.             contour_flag = bpy.props.BoolProperty(
219.                 name="Contours",
220.                 description="Show total contact force contours",
221.                 default=True
222.             )
223.             explosion_flag = bpy.props.BoolProperty(
224.                 name="Explosion",
225.                 description="Activate Explosion",
226.                 default=False
227.             )
228.
229.         class ExplosionPanel(bpy.types.Panel):
230.             bl_idname = "panel.explosion_panel"
231.             bl_label = "Post Processing"
232.             bl_space_type = "VIEW_3D"
233.             bl_region_type = "TOOLS"
234.             bl_category = "BDA"
235.
236.
237.             def draw(self, context):
238.                 layout = self.layout
239.                 scene = context.scene
240.                 explosion_tool = scene.explosion_tool
241.
242.
243.
244.
```

```
245.                # Create an row where the buttons are aligned to each other.
246.                layout.label(text=" Contact Force")
247.                row = layout.row()
248.                row.prop(bpy.context.scene.explosion_tool, "contour_flag")
249.                row = layout.row()
250.                self.layout.operator("scene.debris_log_file_operator")
251.                self.layout.prop(bpy.context.scene.explosion_tool, 'debrisfilepath')
252.
253.                row = layout.row()
254.
255.                self.layout.operator("scene.logfilepath_operator")
256.                self.layout.prop(bpy.context.scene.explosion_tool, 'logfilepath')
257.
258.        class RegisterHandlerOperator(bpy.types.Operator):
259.            bl_idname = "scene.register_handler_operator"
260.            bl_label = "Register Handler"
261.
262.            def execute(self, context):
263.                print('Initializing event handler.')
264.                bpy.app.handlers.frame_change_pre.append(main)
265.                return {'FINISHED'}
266.
267.        class UnregisterHandlerOperator(bpy.types.Operator):
268.            bl_idname = "scene.unregister_handler_operator"
269.            bl_label = "Unregister Handler"
270.
271.            def execute(self, context):
272.                print('Removing event handler.')
273.                for i in range( len( bpy.app.handlers.frame_change_pre ) ):
274.                    bpy.app.handlers.frame_change_pre.pop()
275.                return {'FINISHED'}
276.
277.        class HandlerPanel(bpy.types.Panel):
278.            bl_idname = "panel.handler_panel"
279.            bl_label = "Frame Change Event Handler"
280.            bl_space_type = "VIEW_3D"
281.            bl_region_type = "TOOLS"
282.            bl_category = "BDA"
283.
284.            def draw(self, context):
285.                layout = self.layout
286.                scene = context.scene
287.
288.                self.layout.operator("scene.register_handler_operator")
289.                self.layout.operator("scene.unregister_handler_operator")
290.
291.
292.        class ExplosionOperator (bpy.types.Operator):
293.            bl_idname = "wm.explosion_operator"
294.            bl_label = "Add Source Point"
295.
296.            def execute(self, context):
297.                scene = context.scene
298.                explosion_tool = scene.explosion_tool
299.
300.                i = 0
301.                for item in context.scene.collection:
302.                   print(item.source[0], item.source[1], item.source[2], con-
    text.scene.amountcollection[i].amount)
303.                   i = i + 1
304.                return {'FINISHED'}
```

2.3.3. Contact Force Calculation and Contour Visualization: Theory and Code Development

One of the main features of BDA is its capability of computing the contact forces between fractured bodies during the simulation and visualizing the information with the contours in each time step. Figure 7 shows an example of the contact force contour created by BDA due to the wrecking ball generated impact on the bridge deck of a single span.

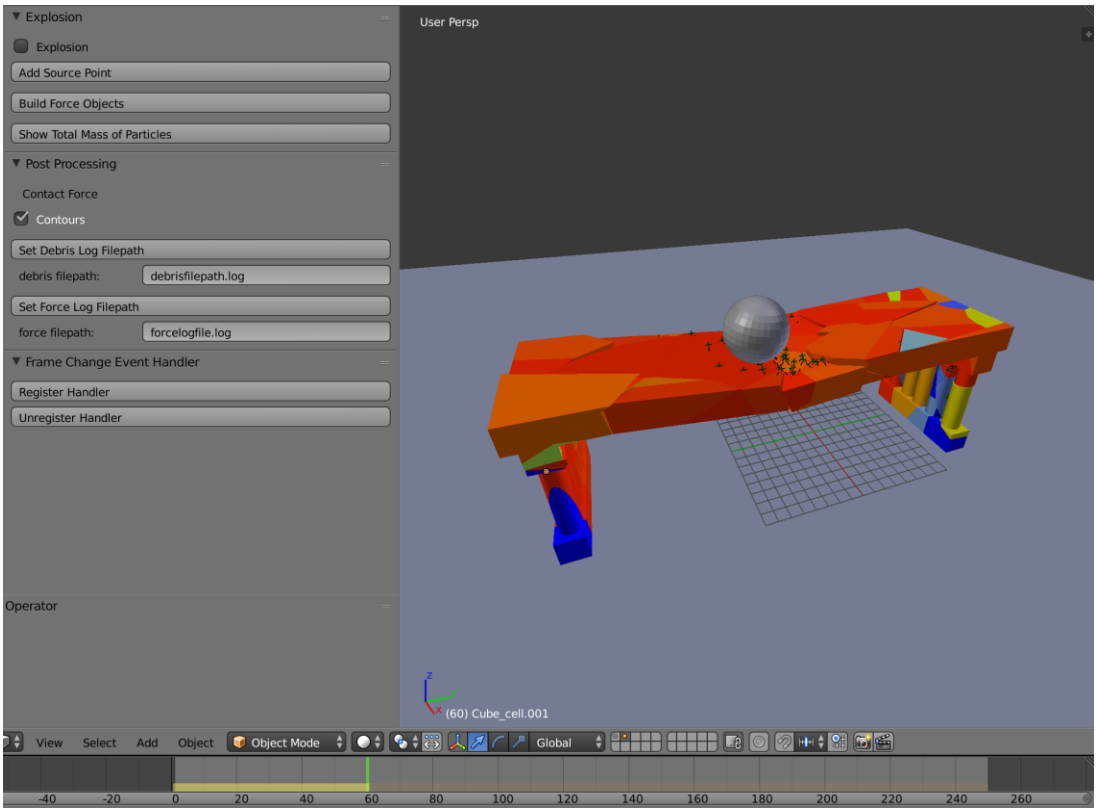

Figure 7. Contact force contour created by BDA; The bridge is under the wrecking ball generated impact

Bullet Physics as the physics engine integrated into Blender, which is an impulse-based simulator that bypasses the contact force computation. However, the contact force is a critical information needed for the engineering applications, for which BDA has been developed to retrieve the lost contact force information. The theory is based on Lee and Hashash [8]. Readers are directed to the reference for the further technical details. The code for contact force calculation and contour visualization has been implemented in Python using Blender's API and is presented here. Some parts of the contour visualization are loosely based on [16].

```
def main(scene):


    if scene.frame_current == scene.frame_end:
        log_debris_locations()
        if bpy.context.screen.is_animation_playing:
            bpy.ops.screen.animation_play()


    if not "start-frame" in bpy.app.driver_namespace:
        em = bpy.app.driver_namespace["start-frame"] = []
        log_debris_locations()


    ############################
    ### What to do on start frame

    if bpy.context.scene.explosion_tool.contour_flag:

            ### Render part
            if qRenderAnimation:

                for i in range( len( bpy.app.handlers.frame_change_pre ) ):
                     bpy.app.handlers.frame_change_pre.pop()

                filepathOld = bpy.context.scene.render.filepath
                bpy.context.scene.render.filepath += "%04d" %(scene.frame_current -1)
                bpy.context.scene.render.image_settings.file_format = 'JPEG'
                bpy.context.scene.render.image_settings.quality = 75


                if qRenderAnimation == 1: bpy.ops.render.render(write_still=True)

                elif qRenderAnimation == 2: bpy.ops.render.opengl(write_still=True)

                bpy.context.scene.render.filepath = filepathOld


                bpy.app.handlers.frame_change_pre.append(main)


                ############################
                ### What to do on start frame
                if not "force-contour" in bpy.app.driver_namespace:
                    print("Initializing buffers...")


                    ###### Function
                    initBuffers(scene)
                    ###### Function
                    initMaterials()
                else:
                    ###### Function
                    ################################
                    ### What to do AFTER start frame
                    print("Frame:", scene.frame_current)

                    defObjs = bpy.app.driver_namespace["force-contour"]

                    ###### Function
```

```
                    changeObjMaterials()


    if bpy.context.scene.explosion_tool.explosion_flag:
        if not "explosion" in bpy.app.driver_namespace:
            print("Initializing buffers...")
            ###### Function
            initExplosion(scene)
        else:

            defObjsExplosion = bpy.app.driver_namespace["explosion"]
            do_explosion(scene.frame_current)


    ################################
    ### What to do AFTER start frame
    print("Frame:", scene.frame_current)




################################################################################

def initExplosion(scene):
    # Create property for frame change storage of data
    defObjsExplosion = bpy.app.driver_namespace["explosion"] = []


################################################################################

def initBuffers(scene):


    ### Make object list
            objs = []
            for obj in bpy.data.objects:
                # exclude constraint objects
                if obj.select and obj.type == 'MESH' and not obj.hide and obj.is_visi-
    ble(scene) and not hasattr(obj.rigid_body_constraint, 'type'):
                    objs.append(obj)

            # Create property for frame change storage of data
            defObjs = bpy.app.driver_namespace["force-contour"] = []
            defObjsDict = bpy.app.driver_namespace["force-contour-dict"] = {}

            # frame force dictionary
            forceDict = {}

            # for all constraints calculate object forces
            objects = bpy.data.objects
            for obj in objects:
                if(hasattr(obj.rigid_body_constraint, 'type')):

                    if obj.rigid_body_constraint.object1.name not in forceDict:
                        forceDict[obj.rigid_body_constraint.object1.name] = 0

                    if obj.rigid_body_constraint.object2.name not in forceDict:
```

```
                        forceDict[obj.rigid_body_constraint.object2.name] = 0


            ### Create original transform data array
            d = -1
            for objname in forceDict:
                d += 1
                if d %100 == 0: sys.stdout.write('\r' +"%d" %d)
                defObjForceDif = 0
                defObjs.append([obj, forceDict[objname], defObjForceDif])
                defObjsDict[objname] = d
                ### Remove all materials but one from object
                objectd = bpy.data.objects[objname]
                bpy.context.scene.objects.active = objectd

                while len(objectd.material_slots) > 1:
                    objectd.active_material_index = 0
                    bpy.ops.object.material_slot_remove()


                objectd["defLastStep"] = 0

            print("defObjsDict", defObjsDict)

        ###############################################################################


        def initMaterials():


            ### Create gradient materials for later use and reuse
            for step in range(displaySteps):
                dif = step *(1 /(displaySteps -1))

                col = Color((0, 0, 0))
                #col.h = 0; col.s = 1; col.v = 1
                difNormal = dif
                col.r = difNormal   # correct math: = (difNormal -0.5) *2
                col.g = 1 -(abs(0.5 -difNormal) *2)
                col.b = (0.5 -difNormal) *2

                mat = bpy.data.materials.new(materialName +"%03d" %step)
                mat.diffuse_color = col
                col.s *= 0.5
                mat.specular_color = col
                mat.emit = 0.33


        ###############################################################################


        def changeObjMaterials():


            time_scale = bpy.context.scene.rigidbody_world.time_scale
            steps_per_second = bpy.context.scene.rigidbody_world.steps_per_second

            # Get property data
```

```
181.            defObjs = bpy.app.driver_namespace["force-contour"]
182.            defObjsDict = bpy.app.driver_namespace["force-contour-dict"]
183.
184.            # frame force dictionary
185.            forceDict = {}
186.            # for all constraints calculate object forces
187.            objects = bpy.data.objects
188.            for obj in objects:
189.                if(hasattr(obj.rigid_body_constraint, 'type')):
190.                    impulse = obj.rigid_body_constraint.appliedImpulse()
191.                    force = impulse / time_scale * steps_per_second
192.
193.                    if obj.rigid_body_constraint.object1.name in forceDict:
194.                        forceDict[obj.rigid_body_constraint.object1.name] += force
195.                    else:
196.                        forceDict[obj.rigid_body_constraint.object1.name] = force
197.
198.                    if obj.rigid_body_constraint.object2.name in forceDict:
199.                        forceDict[obj.rigid_body_constraint.object2.name] -= force
200.                    else:
201.                        forceDict[obj.rigid_body_constraint.object2.name] = 0 - force
202.
203.
204.
205.            if (not bpy.context.scene.explosion_tool.logfilepath):
206.                bpy.context.scene.explosion_tool.logfilepath = "forcelogfile.log"
207.            f = open(bpy.context.scene.explosion_tool.logfilepath, "a+")
208.            listForces = []
209.            for defObj in forceDict:
210.                f.write(str(defObj) + ":" + str(forceDict[defObj]) + "\n")
211.                listForces.append(forceDict[defObj])
212.
213.            minf = 0.0
214.            maxf = 0.0
215.
216.            for li in listForces:
217.                if li < minf:
218.                    minf = li
219.
220.                if li > maxf:
221.                    maxf = li
222.
223.
224.
225.            d = -1
226.            for defObj in forceDict:
227.                d += 1
228.                if d %100 == 0: sys.stdout.write('\r' +"%d" %d)
229.                objectd = bpy.data.objects[defObj]
230.                defObjForce = defObjs[defObjsDict[defObj]][1] # old force
231.
232.                # print("old force: ", defObjForce)
233.                # f.write("old force" + str(defObjForce) + ",")
234.
235.                ### Find material
236.                objForce = forceDict[defObj]   # new force
237.
238.                # print("new force: ", objForce)
239.                # f.write("new force: " + str(objForce) + ",")
240.
241.                # Calculate deformation differences
```

```
242.                defObjForceDif = defObjForce - objForce
243.
244.                defObjs[defObjsDict[defObj]][1] = objForce # set new force
245.                defObjs[defObjsDict[defObj]][2] = defObjForceDif # set new forceDif
246.
247.                # print("defObjForceDif: ", defObjForceDif)
248.                # f.write("defObjForceDif: " + str(defObjForceDif) + ",")
249.
250.
251.                dif = abs(defObjForceDif)
252.
253.                diff = dif /2000
254.
255.                if diff < displaySteps: step = int(diff)
256.                else: step = displaySteps -1
257.
258.
259.
260.                stepLast = objectd["defLastStep"]
261.                if step < stepLast -fadeDecrement: step = stepLast -fadeDecrement
262.
263.                mat = bpy.data.materials[materialName +"%03d" %step]
264.                objectd["defLastStep"] = step
265.
266.
267.
268.                objectd.material_slots[-1:][0].material = mat
```

2.3.4. Implementation of capability to simulate the demolition by explosion: Theories, Formulations and Code Development

Explosion is one of the major methods of bridge demolition and one of the major features implemented in BDA. Figure 8 shows an example of simulation of demolition by explosives with BDA.

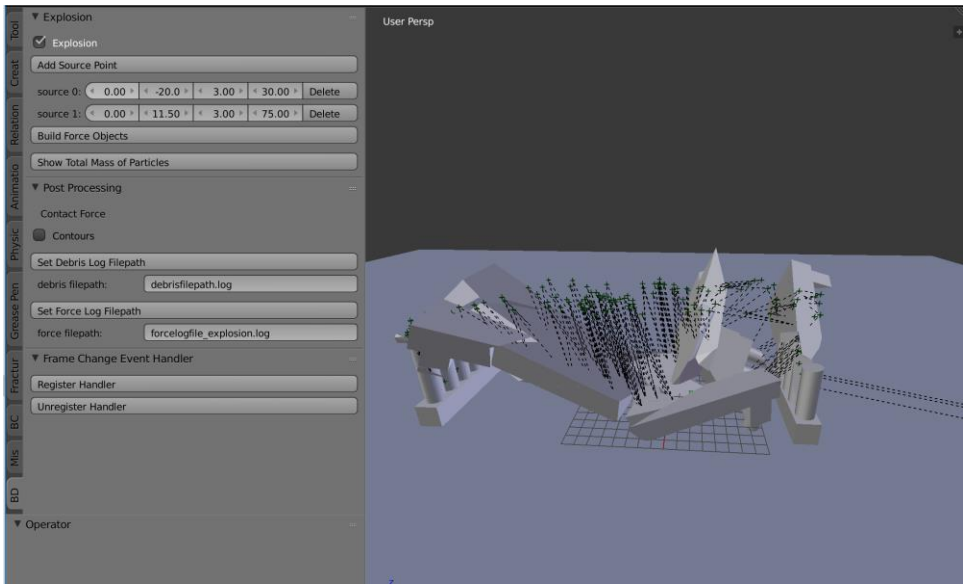

Figure 8. An explosion simulation performed by Bridge Demolition Add-on for a single span bridge

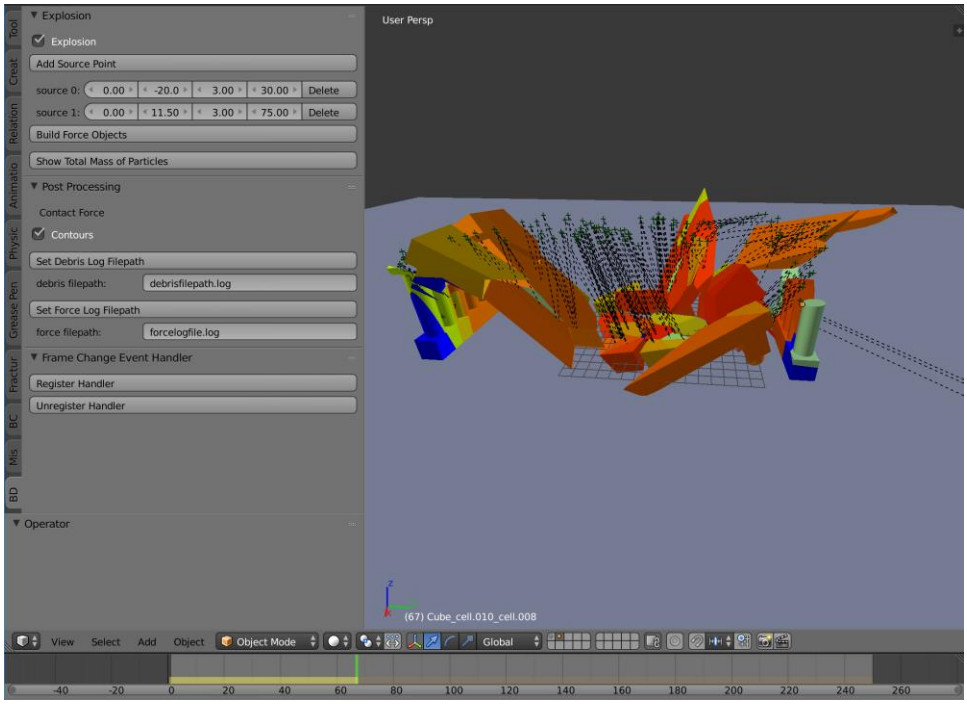

Figure 9. Contact force contour visualization by the explosives

Explosion is a sudden energy release which generates a blast wave. The blast wave is assumed to show a spherical propagation with a pressure profile. Below is some key formula of the blast wave theory by explosion. Interested readers are referred to [17] for more reading.

The pressure profile generated by explosion is defined by Friedlander equation:

$$p(t) = p^0(1 - \frac{t}{t_d})e^{-\alpha t/t_d} \quad (1)$$

where p is the instantaneous overpressure at time t; p^0 is the maximum or peak overpressure observed when t is zero at the moment of explosion; t_d is the time duration; and α is a constant [17].

The corresponding force on the surrounding bodies can be computed from the pressure with the following equation:

$$\vec{F} = p(t)\, A\, \frac{\vec{R}}{\|\vec{R}\|} \quad (2)$$

where $p(t)$ is the pressure generated by explosion; A is the projected area of a body; and $\vec{R}$ is the position vector that indicates the position of the body from source of explosion.

The overpressure vs. distance relation for an explosion can be shown by the following equation:

$$\frac{p^0}{P_a} = \frac{808 * [\,1 + \left(\frac{Z}{4.5}\right)^2\,]}{\sqrt{1 + (\frac{Z}{0.048})^2} * \sqrt{1 + \left(\frac{Z}{0.32}\right)^2} * \sqrt{1 + \left(\frac{Z}{1.35}\right)^2}} \quad (3)$$

where $\frac{p^0}{P_a}$ is the ratio of peak overpressure to ambient atmospheric pressure; and Z is called scaled-distance that is the actual distance in meters [17].

The duration, t_d, is the time duration in which explosive forces are applied and hence is an index of how damaging the explosion can be. It's common to assume the positive overpressure phase as the time duration of the explosion:

$$\frac{t_d}{W^{\frac{1}{3}}} = \frac{980 * [\,1 + \left(\frac{Z}{0.54}\right)^{10}\,]}{\left[1 + (\frac{Z}{0.02})^3\right] * \left[1 + (\frac{Z}{0.74})^6\right] * \sqrt{1 + (\frac{Z}{6.9})^2}} \quad (4)$$

where $\frac{t_d}{W^{\frac{1}{3}}}$ is the explosion duration time in milliseconds for one kilogram TNT explosion and Z is the scaled distance in meters; and t_d is the time duration of explosive forces.

The scaling law for explosions yields as below based on conservation of momentum and geometric similarity:

$$D_s = \frac{f_d * D_a}{W^{\frac{1}{3}}} \quad (5)$$

where D_s is the scaled distance to the source of explosion; D_a is the actual distance to the source of explosion; and f_d is the transmission factor which is assumed to be equal to 1, which means that the change of atmospheric density to due to explosion is ignored.

Figure 10 shows how overpressure ratio varies with time for different scaled distances generated by a W equal to 10,000 kg of TNT. Figure 11 shows how overpressure ratio due to explosion varies by scaled distance for different explosion intensities. Explosion intensity is in terms of W in kilograms of TNT. These examples of contour are created by the MATLAB code implemented as a part of this project. With use of these codes, users can estimate the expected overpressure ratio

in a specific distance from explosion source point with respect to explosion intensity applied. Figure 12 shows an example regarding how much explosion intensity is required given a target overpressure (i.e. a desired amount of damage) to be applied in a specific distance.

As a real example, the demolition of a two-span bridge (each span has a length of 20 meters) is considered. Total amount of 1000 kg of explosives is installed at both ends and mid piers of the bridge. Figure 13 shows how overpressure ratio varies by scaled distance due to explosion at each pier and Figure 14 shows the total overpressure ratio along the bridge due to the explosion.

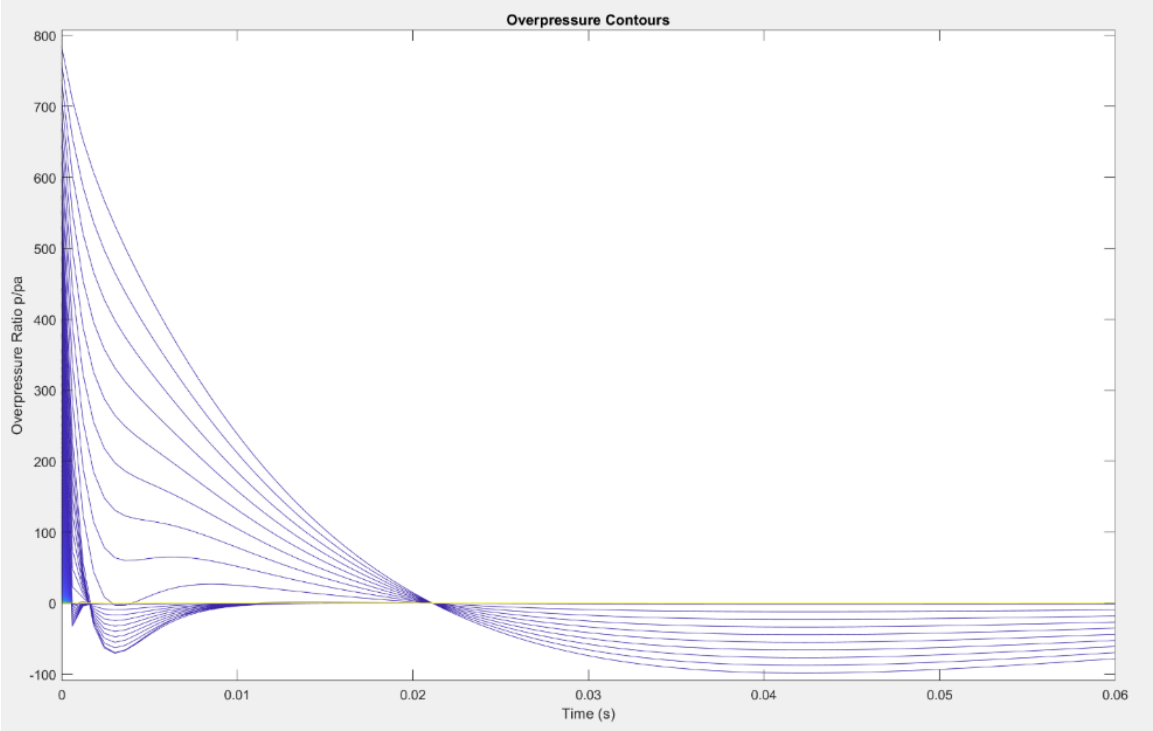

Figure 10. Variation of overpressure ratio vs time for different scaled distances

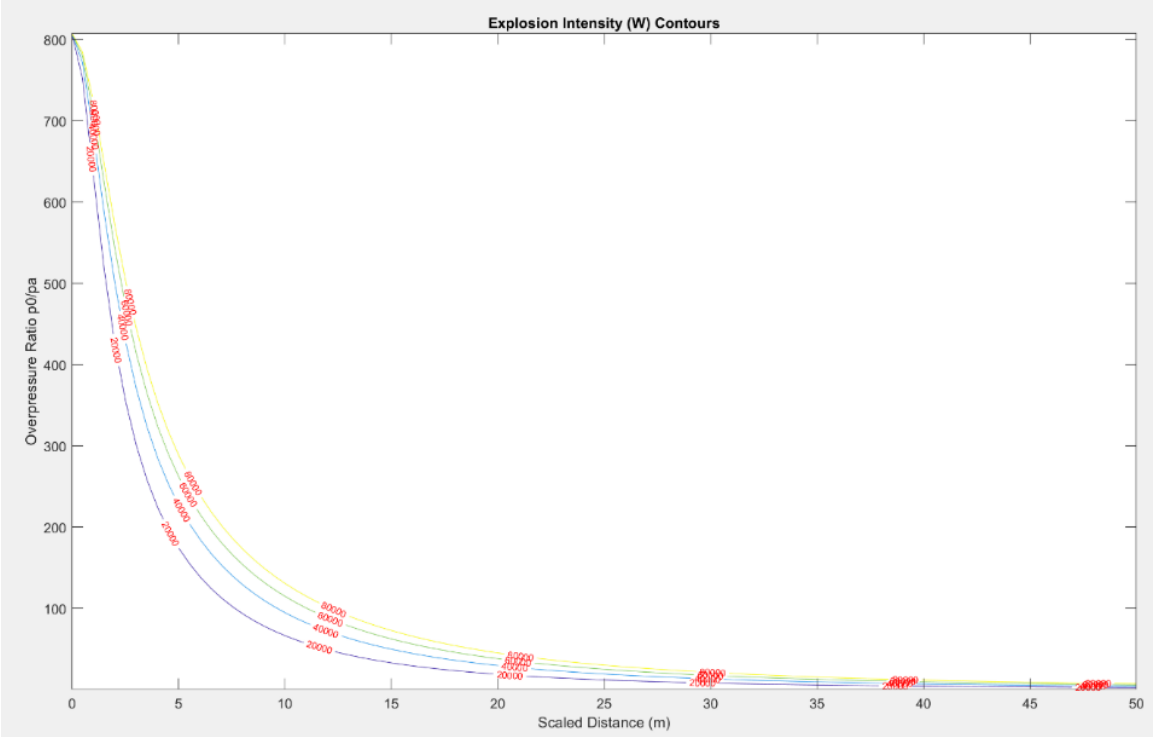

Figure 11. Explosion intensity contours

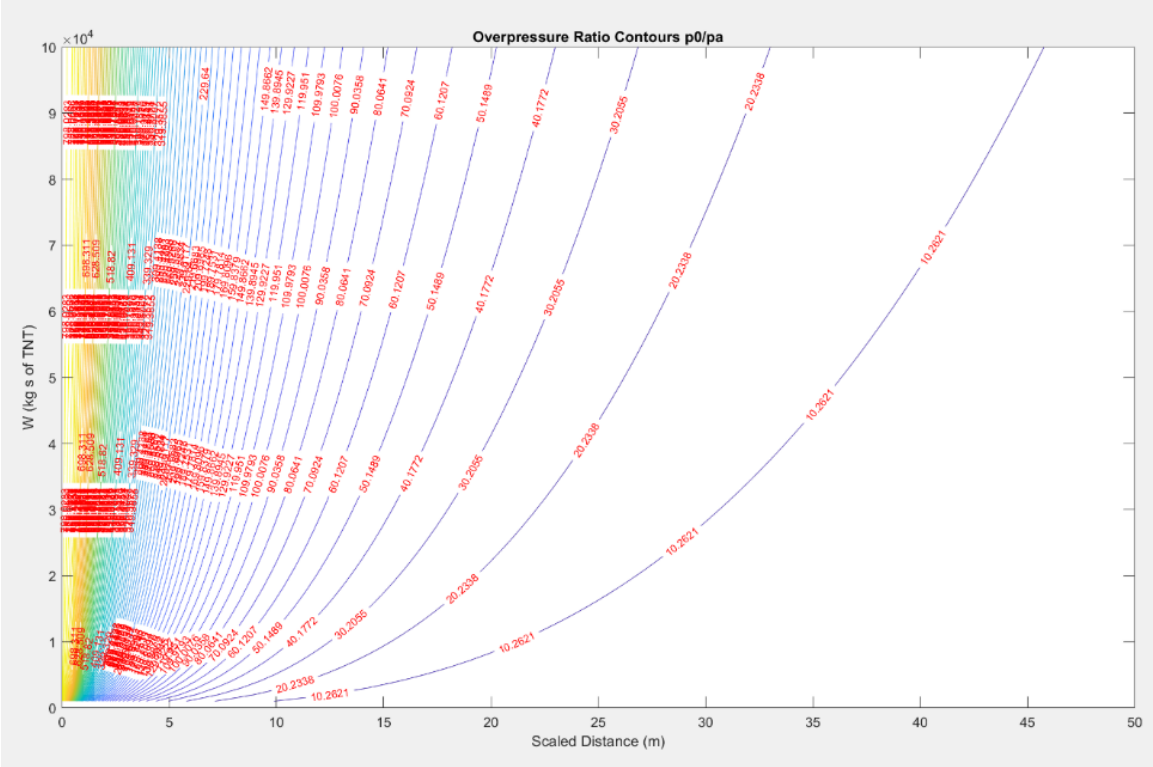

Figure 12. Overpressure ratio contours

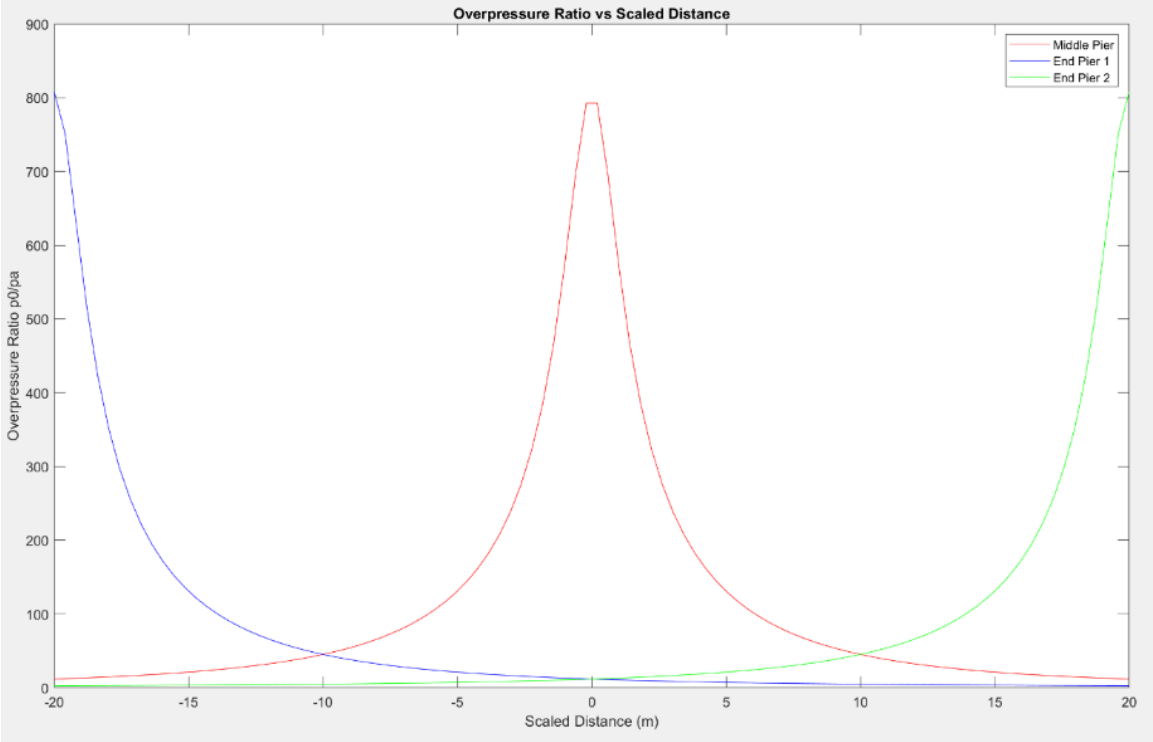

Figure 13. Variation of overpressure vs. scaled distance for a 2-span bridge exploded by 1 ton of explosives

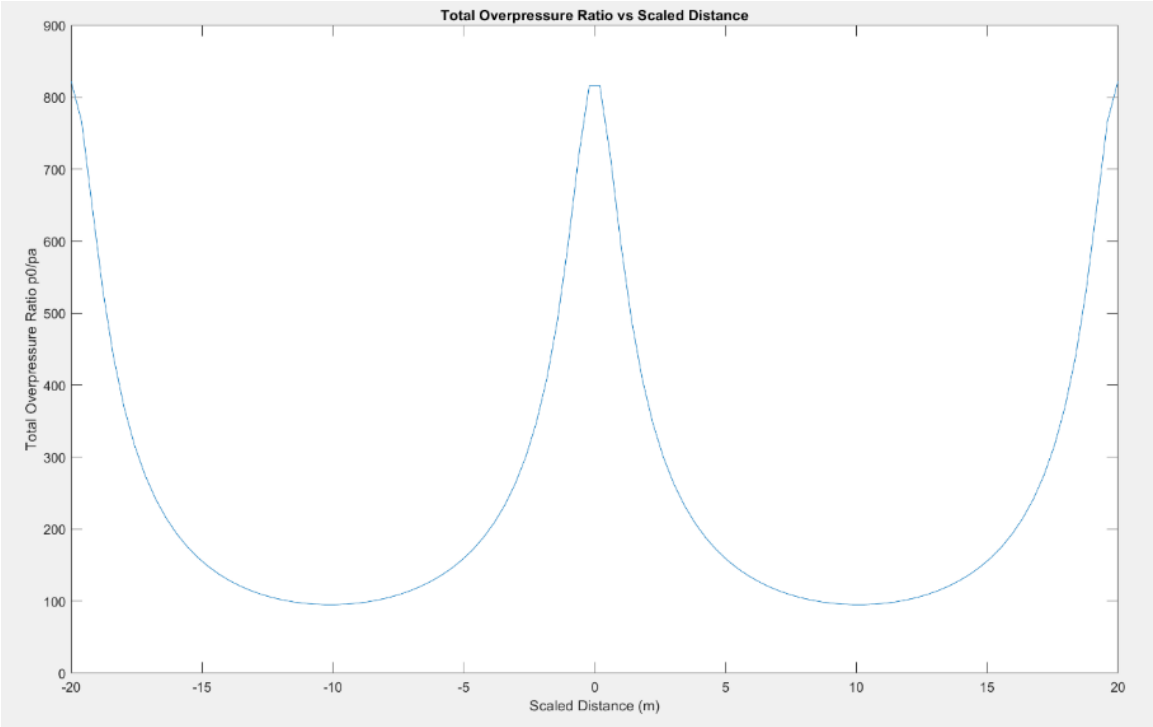

Figure 14. Variation of total overpressure vs. scaled distance for a 2 span bridge exploded by 1 ton of explosives

One of the main challenges in developing code for implementing the explosion is the calculation of projected areas of each fractured shard with regards to explosion induced pressure. The convex hull is computed from the discretized shards' geometry, which is then used to estimate the explosion force applied to the body. BDA is equipped with the functions that can precisely calculate the projected area of convex hull and update the information each time step with respect to the updated locations and orientations of the fractured bodies. Projected area of a convex hull with respect to a designated explosion source point is calculated as in Equation:

$$PA = \frac{1}{2}\sum_{i=1}^{k}(|\hat{n}.\hat{n_i}|S_i) \tag{6}$$

where PA is the projected area of the convex hull; k is the number of faces of a convex hull; $\hat{n}$ is normal vector of the projection plane (which is simply the subtraction of fracture center of mass location vector and explosion source point location vector); and $\hat{n_i}$ is the normal vector of the face; and S_i is the face area.

Projected area computation is coded in Python using Blender's API as follows:

```
1.  ######################################################################################
2.  # bmesh from object
3.
4.  def bmesh_from_object(ob):
5.      matrix = ob.matrix_world
6.      me = ob.to_mesh(bpy.data.scenes['Scene'], apply_modifiers=True, settings='PRE-
    VIEW')
7.      me.transform(matrix)
8.      bm = bmesh.new()
9.      bm.from_mesh(me)
10.     bpy.data.meshes.remove(me)
11.     return bm
12.
13. ######################################################################################
14. # dot product
15.
16. def dot_product(vec1, vec2):
17.     sum = vec1[0]*vec2[0]+vec1[1]*vec2[1]+vec1[2]*vec2[2]
18.     return sum
19.
20. ######################################################################################
21. # projected area
22.
23. def projected_area(obj, projection_normal_vector):
24.     bm = bmesh_from_object(obj)
25.     bm.normal_update()
26.     print('bmesh:', bm)
27.
28.     pa = 0
29.     for f in bm.faces:
```

```
        face_area = f.calc_area()
        face_normal = f.normal

        # compute projected area
        face_pa = abs(dot_product(projection_normal_vector, face_normal)) * face_area
        pa += face_pa
        print('face index:', f.index, ' and face area:', face_area, 'face normal:', face_normal, 'face pa:', face_pa)
        print

    bm.free()
    pa = pa/2
    return pa


################################################################################
```

Explosion force applied to any fracture from a source point is calculated by the function implemented as follows:

```
def explosion(m_origin, m_source, time_elapsed, projected_area, amount):
    Distance = [0,0,0] #btVector3

    w = amount;

    Distance[0] = m_origin[0] - m_source[0]
    Distance[1] = m_origin[1] - m_source[1]
    Distance[2] = m_origin[2] - m_source[2]

    distance = pow((Distance[0] * Distance[0] + Distance[1] * Distance[1] + Distance[2] * Distance[2]), 0.5);

    distance /= pow(w,1/3);


    area = projected_area;

    ambientPressure = 101325.0;

    peakOverpressure = ambientPressure * 808 * ((pow(distance / 4.5, 2) + 1) / (pow((1 + (pow(distance / 0.048, 2))), 0.5) * pow((1 + (pow(distance / 0.32, 2))), 0.5) * pow((1 + (pow(distance / 1.35, 2))), 0.5)));

    millisecond_to_second = 0.001

    td = millisecond_to_second * 980 * (1 + pow(distance / 0.54, 10) / ((1 + pow(distance / 0.02, 3.0)) * (1 + pow(distance / 0.74, 6)) * (pow(1 + pow(distance / 6.9, 2), 0.5))) * pow(w, 1/3));

    beta = -1.0;

    #Friedlander Equation
    pressure = ambientPressure + peakOverpressure * (1 - time_elapsed / td) * math.exp(beta * time_elapsed / td);

```

```
    explosionforcex = pressure * area * Distance[0] / pow((Distance[0] * Distance[0] + Distance[1] * Distance[1] + Distance[2] * Distance[2]), 0.5);
    explosionforcey = pressure * area * Distance[1] / pow((Distance[0] * Distance[0] + Distance[1] * Distance[1] + Distance[2] * Distance[2]), 0.5);
    explosionforcez = pressure * area * Distance[2] / pow((Distance[0] * Distance[0] + Distance[1] * Distance[1] + Distance[2] * Distance[2]), 0.5);

    m_explosionForce = [explosionforcex, explosionforcey, explosionforcez];
    return m_explosionForce;
```

Code development of explosion implementation in Blender's API is as follows, where the Force field in Blender has been leveraged:

```
# Field Force Properties
FIELD_HIDE = True
FIELD_TYPE = 'FORCE'
FIELD_Z_DIRECTION = 'BOTH'
FIELD_SHAPE = 'POINT' #'PLANE'
FIELD_STRENGTH = 0.0
FIELD_FLOW = 0.0
FIELD_SEED = 40
FIELD_APPLY_TO_LOCATION = True
FIELD_APPLY_TO_ROTATION = True
FIELD_FALLOFF_TYPE = 'TUBE'
FIELD_FALLOFF_POWER = 10.0
FIELD_USE_MIN_DISTANCE = True
FIELD_USE_MAX_DISTANCE = False
FIELD_DISTANCE_MAX = 5 # 0.2
FIELD_USE_RADIAL_MIN = False
FIELD_USE_RADIAL_MAX = True
FIELD_RADIAL_MAX = 2 # 0.2
FIELD_RADIAL_FALLOFF = 0.0 # real graviational falloff = 2!!

# Rotation
ROTATION_90 = 1.5707961320877075


class ExplosionOperator (bpy.types.Operator):
    bl_idname = "wm.explosion_operator"
    bl_label = "Add Source Point"

    def execute(self, context):
        scene = context.scene
        explosion_tool = scene.explosion_tool

        i = 0
        for item in context.scene.collection:
           print(item.source[0], item.source[1], item.source[2], context.scene.amountcollection[i].amount)
           i = i + 1
        return {'FINISHED'}


def build_all_force_objects():
    for obj in bpy.data.objects:
        if obj.type != 'MESH':
            continue

```

```
        if not hasattr(obj.rigid_body, 'type'):
            continue

        if obj.rigid_body.type != 'ACTIVE':
            continue

        # create_empty_obj
        create_empty_obj(obj, 0)
        create_empty_obj(obj, 1)
        create_empty_obj(obj, 2)

################################################################################
# create empty object

def create_empty_obj(parent_obj, axes_index):
    emptyname = 'Empty_' + str(axes_index) + '_' + parent_obj.name

    if bpy.data.objects.get(emptyname) is None:
        bpy.ops.object.empty_add(type='PLAIN_AXES')
        emptyobj = bpy.data.objects[bpy.context.active_object.name]
        emptyobj.name = emptyname
        emptyobj.select = False
        emptyobj.hide = FIELD_HIDE
        #emptyobj.parent = parent_obj
        emptyobj.location = parent_obj.location
        emptyobj.rotation_euler[0] = 0
        emptyobj.rotation_euler[1] = 0
        emptyobj.rotation_euler[2] = 0
        if (axes_index < 2):
            emptyobj.rotation_euler[axes_index] = ROTATION_90

        emptyobj.field.type = FIELD_TYPE
        emptyobj.field.z_direction = FIELD_Z_DIRECTION
        emptyobj.field.shape = FIELD_SHAPE
        emptyobj.field.strength = FIELD_STRENGTH
        emptyobj.field.flow = FIELD_FLOW
        emptyobj.field.seed = FIELD_SEED
        emptyobj.field.apply_to_location = FIELD_APPLY_TO_LOCATION
        emptyobj.field.apply_to_rotation = FIELD_APPLY_TO_ROTATION
        emptyobj.field.falloff_type = FIELD_FALLOFF_TYPE
        emptyobj.field.falloff_power = FIELD_FALLOFF_POWER
        emptyobj.field.use_min_distance = FIELD_USE_MIN_DISTANCE
        emptyobj.field.use_max_distance = FIELD_USE_MAX_DISTANCE
        emptyobj.field.distance_max = FIELD_DISTANCE_MAX
        emptyobj.field.use_radial_min = FIELD_USE_RADIAL_MIN
        emptyobj.field.use_radial_max = FIELD_USE_RADIAL_MAX
        emptyobj.field.radial_max = FIELD_RADIAL_MAX
        emptyobj.field.radial_falloff = FIELD_RADIAL_FALLOFF


################################################################################
def update_empty_objects(parent_obj, explosion_force):
    for i in range(3):
        emptyobj = bpy.data.objects["Empty_" + str(i) + "_" + parent_obj.name]
        emptyobj.location = parent_obj.location
                #emptyobj.field.strength = explosion_force[i]
                if i < 2:
                    emptyobj.field.strength = explosion_force[abs(i-1)]
                else :
                    emptyobj.field.strength = explosion_force[i]

```

```
106.
107.       ###############################################################################

108.
109.       # explosion
110.
111.       def explosion(m_origin, m_source, time_elapsed, projected_area, amount):
112.           Distance = [0,0,0] #btVector3
113.
114.           w = amount;
115.
116.           Distance[0] = m_origin[0] - m_source[0]
117.           Distance[1] = m_origin[1] - m_source[1]
118.           Distance[2] = m_origin[2] - m_source[2]
119.
120.           distance = pow((Distance[0] * Distance[0] + Distance[1] * Distance[1] + Dis-
    tance[2] * Distance[2]), 0.5);
121.
122.           distance /= pow(w,1/3);
123.
124.
125.           area = projected_area;
126.
127.           ambientPressure = 101325.0;
128.
129.           peakOverpressure = ambientPressure * 808 * ((pow(dis-
    tance / 4.5, 2) + 1) / (pow((1 + (pow(distance / 0.048, 2))), 0.5) * pow((1 + (pow(dis-
    tance / 0.32, 2))), 0.5) * pow((1 + (pow(distance / 1.35, 2))), 0.5)));
130.
131.           millisecond_to_second = 0.001
132.
133.           td = millisecond_to_second * 980 * (1 + pow(dis-
    tance / 0.54, 10) / ((1 + pow(distance / 0.02, 3.0)) * (1 + pow(dis-
    tance / 0.74, 6)) * (pow(1 + pow(distance / 6.9, 2), 0.5))) * pow(w, 1/3));
134.
135.           beta = -1.0;
136.
137.           #Friedlander Equation
138.           pressure = ambientPressure + peakOverpres-
    sure * (1 - time_elapsed / td) * math.exp(beta * time_elapsed / td);
139.
140.           explosionforcex = pressure * area * Distance[0] / pow((Distance[0] * Dis-
    tance[0] + Distance[1] * Distance[1] + Distance[2] * Distance[2]), 0.5);
141.           explosionforcey = pressure * area * Distance[1] / pow((Distance[0] * Dis-
    tance[0] + Distance[1] * Distance[1] + Distance[2] * Distance[2]), 0.5);
142.           explosionforcez = pressure * area * Distance[2] / pow((Distance[0] * Dis-
    tance[0] + Distance[1] * Distance[1] + Distance[2] * Distance[2]), 0.5);
143.
144.           m_explosionForce = [explosionforcex, explosionforcey, explosionforcez];
145.           return m_explosionForce;
146.
147.
148.
149.       ###############################################################################

150.       # bmesh from object
151.
152.       def bmesh_from_object(ob):
153.           matrix = ob.matrix_world
154.           me = ob.to_mesh(bpy.data.scenes['Scene'], apply_modifiers=True, set-
    tings='PREVIEW')
```

```
155.            me.transform(matrix)
156.            bm = bmesh.new()
157.            bm.from_mesh(me)
158.            bpy.data.meshes.remove(me)
159.            return bm
160.
161.        ###################################################################################

162.        # dot product
163.
164.        def dot_product(vec1, vec2):
165.            sum = vec1[0]*vec2[0]+vec1[1]*vec2[1]+vec1[2]*vec2[2]
166.            return sum
167.
168.        ###################################################################################

169.        # projected area
170.
171.        def projected_area(obj, projection_normal_vector):
172.            bm = bmesh_from_object(obj)
173.            bm.normal_update()
174.            print('bmesh:', bm)
175.
176.            pa = 0
177.            for f in bm.faces:
178.                face_area = f.calc_area()
179.                face_normal = f.normal
180.
181.                # compute projected area
182.                face_pa = abs(dot_product(projection_normal_vector, face_nor-
    mal)) * face_area
183.                pa += face_pa
184.                print('face index:', f.index, ' and face area:', face_area, 'face nor-
    mal:', face_normal, 'face pa:', face_pa)
185.                print
186.
187.            bm.free()
188.            pa = pa/2
189.            return pa
190.
191.
192.
193.
194.        ###################################################################################

195.
196.        def do_explosion(current_frame):
197.            print("do:", current_frame)
198.            # deselect all of the objects
199.            bpy.ops.object.select_all(action='DESELECT')
200.
201.            time_scale = bpy.context.scene.rigidbody_world.time_scale
202.            steps_per_second = bpy.context.scene.rigidbody_world.steps_per_second
203.
204.            time_elapsed = current_frame * time_scale / steps_per_second
205.
206.            if (not bpy.context.scene.explosion_tool.logfilepath):
207.                bpy.context.scene.explosion_tool.logfilepath = "forcelogfile_explo-
    sion.log"
208.            f = open(bpy.context.scene.explosion_tool.logfilepath+"_explo-
    sion.log", "a+")
```

```
            f.write('name, projected area, explosion force, projection_normal_vec-
    tor\n')
            f.write('current_frame: ' + str(cur-
    rent_frame) + 'elapsed time: ' + str(time_elapsed) + '\n')
            for obj in bpy.data.objects:
                if obj.type != 'MESH':
                    continue

                if not hasattr(obj.rigid_body, 'type'):
                    continue

                if obj.rigid_body.type != 'ACTIVE':
                    continue

                projection_normal_vector = obj.rigid_body.location - SOURCE_POINT

                pa = projected_area(obj, projection_normal_vector)

                index = 0
                total_explosion_force = [0, 0, 0]
                for item in bpy.context.scene.collection:
                    amount = bpy.context.scene.amountcollection[index].amount
                    print("amount", amount)
                    explosion_force = explosion(obj.rigid_body.loca-
    tion, item.source, time_elapsed, pa, amount)
                    print("explosion_force: ", explosion_force)
                    for i in range(3):
                        total_explosion_force[i] = total_explosion_force[i] + explo-
    sion_force[i]
                        print("total_explosion_force: ", total_explosion_force)
                    index = index +1

                print('name:', obj.name, 'projected area:', pa, 'explosion force:', to-
    tal_explosion_force, 'projection_normal_vector:', projection_normal_vector)
                f.write(obj.name + ',' + str(pa) + ',' + str(explo-
    sion_force) + ',' + str(projection_normal_vector) + '\n')

                # apply force
                update_empty_objects(obj, total_explosion_force)

            f.close()
```

2.4. Bullet Physics Overview and Basic Concepts

2.4.1. Bullet Physics Overview

Bullet Physics is an open source and free library written in C++, which is widely used as physics engine on various platforms including Play Station 3, Xbox 360, PC, Linux, Mac OSX, Android and iPhone for developing games. Bullet Physics is under zlib license, and its modification can be freely redistributed as far as the terms in the public license are agreed. Interested readers are directed to [18] for further details.

2.4.2. Bullet Basic Data Types and Math Library

The basic data types in Bullet are *btScalar*, *btVector3*, *btQuaternion* and *btMatrix3X3* and *btTransform*. These types along with memory management and containers can be found in *Bullet/src/LinearMath*.

btScalar data type is used for compiling the library in single-floating point precision and double precision. It is a typedef to float and can be also used for double precision arithmetic.

btVector3 is defined for indicating vectors and 3D positions. It includes three scalar components x, y, and z, and an extra component for considering alignment and single instruction multiple data compatibility. A variety of operations such as adding, subtracting and cross product of two vectors can be done with it.

btQuaternion and *btMatrix3X3* are defined and used for showing 3D orientations and rotations.

btTransform is used for transferring vectors between different coordinates and is a container of position and orientation.

These types along with memory management and containers can be found in *Bullet/src/LinearMath*.

2.4.3. Collision Detection

Collision detection is one of the most important and computationally expensive part of the physically based engineering simulation. It is generally desired to obtain collision impulse produced between the pair of polygonal rigid bodies in collision, prevent them penetrating each other. Bullet splits contact detection to two phases: Broad Phase and Narrow Phase. In simulation two rigid

bodies are defined as colliding each other when the distance between their centers of mass is within a tolerance (broad phase) or their geometries are intersecting (narrow phase) [19].

The broad phase is to consider the pairs of rigid bodies possibly collide in the narrow phase and exclude the pairs that do not actually collide. The broad phase uses a bounding shape and a space partitioning to generate an upper bound list of colliding pairs. Various bounding shapes can be considered such as Axis-Aligned Bounding Boxes (AABBs), Oriented Bounding Boxes (OBBs) or simply bounding circle (or sphere). Using these bounding shapes enables the contact detection computationally more manageable by quickly culling out the pairs of not potentially colliding each other. When the AABBs intersect, then the pair is eligible for next narrow phase contact detection, otherwise the pair is excluded.

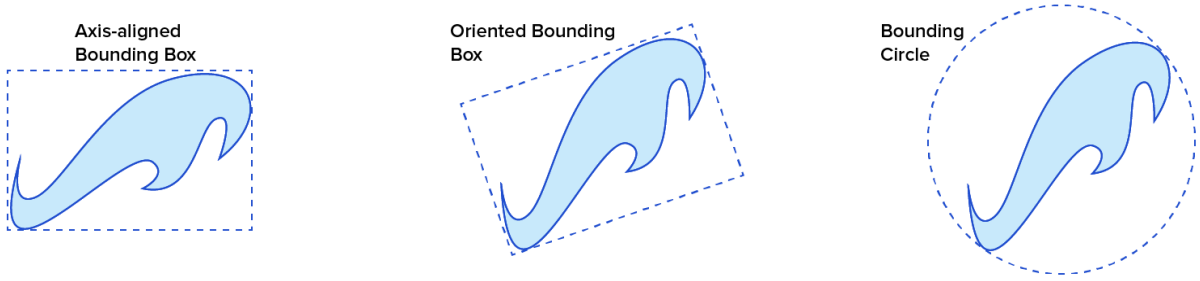

Figure 15. Most used bounding shapes. [19]

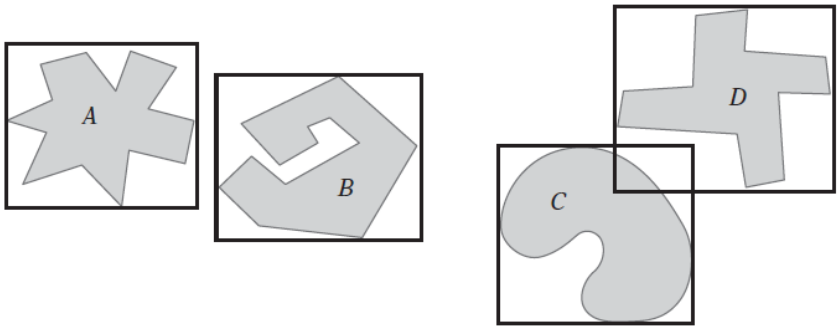

Figure 16. AABBs used for Broad Phase. [20]

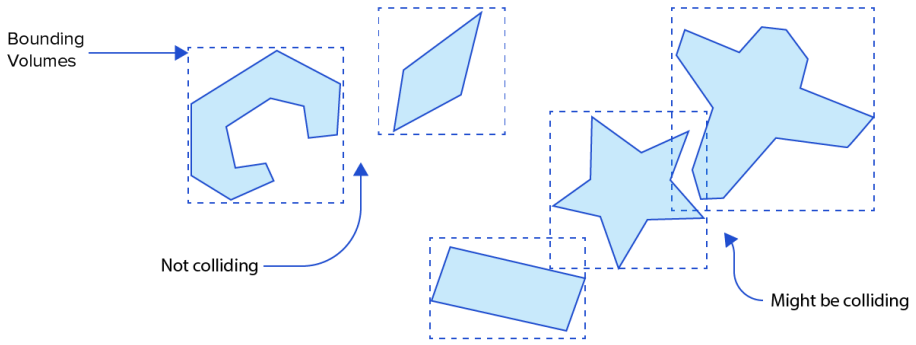

Figure 17. Broad Phase using AABBs [19]

Performing detection for all *n* pairs in the world make the computational order of O(n^2) which is a bottleneck. So broad phase uses space partitioning algorithms such as sort and sweep, and bounding volume hierarchies (BVH). Bullet has the implementation for both in *btAxisSweep* class and *btDvbtBroadphase* class. Dvbt stands for dynamic bounding volume trees which is a type of BVH. The concept of sort and sweep algorithm is to project all of bounding shapes of bodies in the world onto a coordinate axis. This projection will generate intervals for each AABB on the axis, and the algorithm uses the beginning-end intervals to detect intersection. This procedure considerably reduces the number of direct intersection tests [20].

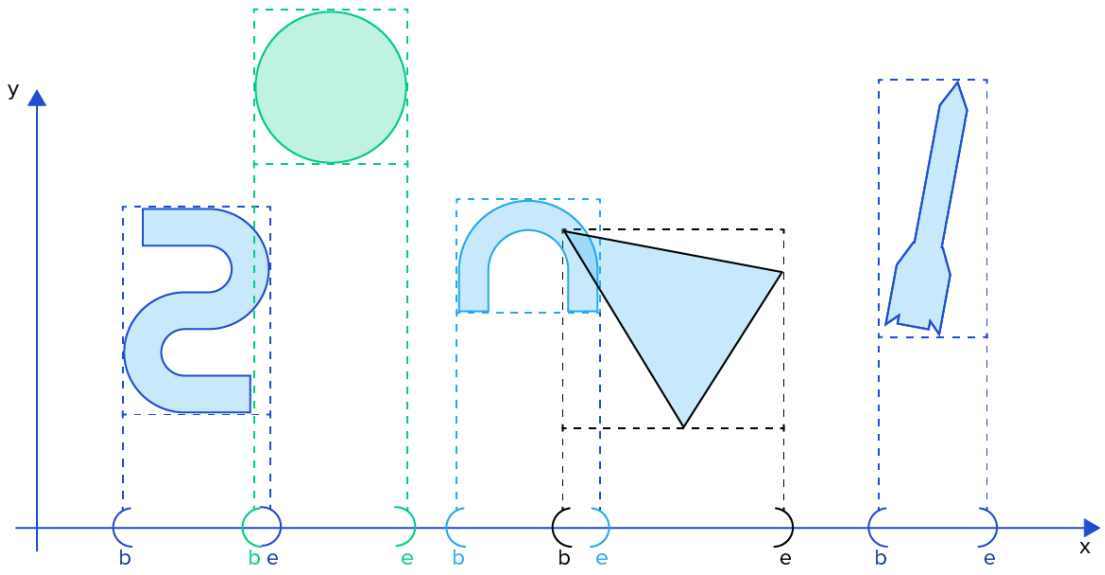

Figure 18. Sort and sweep algorithm [19]

Any interval as [b, e] and sort all their scalar values of b and e ascending is defined in an array. Then by travelling across the array, whenever a b value is faced, the related interval will be sent to the intersecting intervals list and whenever an e value is faced, the related interval is removed from the intersecting list. Bullet uses three structures for broad phase which are *btDbvtBroadphase*, *btAxisSweep3* and *btSimpleBroadphase* [18].

The broad phase finds the colliding pairs and passes a list of potentially colliding pairs passed to the narrow phase each timestep. Narrow phase then conducts a series of detailed contact detection using the actual geometry of the bodies and determines if the pair do collide or not. If they do collide, this procedure identifies the contact points and pass the information to the collision solver, where the collision impulse is computed to resolve the collision.

If the distance of two potentially colliding objects is less than a predefined tolerance, then the pair is actually colliding. The major computational challenge is to compute the shortest distance between the two colliding objects. It is generally prefered to work with convex shapes rather than concave shapes due to the relative simplicity of shortest distance computation, while the concave shape can be decomposed of a set of convex shapes.

Separating Axis Theorem (SAT) is one of the widely used algorithms in the narrow phase contact detection. The 3D version of this theorem is separating plane theorem or SAP. Based on SAT, if and only if there is a single axis that the orthographical projections of two convex shapes don't intersect, then the two shapes are not intersecting each other.

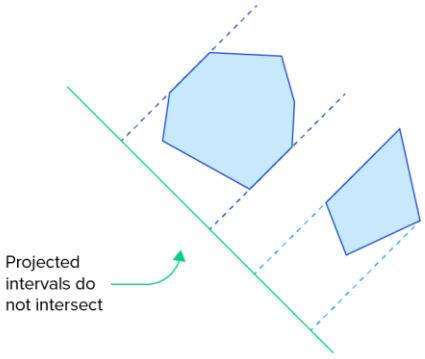

Figure 19. SAT algorithm [19]

Gilbert-Johnson-Keerthi (GJK) is another narrow phase algorithm, which uses Minkowski difference to determine the distance between two shapes and searches for the closest point to the origin

in the resulting shape in an iterative manner. Bullet Physics is also equipped with the implementated GJK algorithm in *btGjkEpaSolver* and *btGjkPairDetector* classes.

Bullet has four major data structures for collision detection: *btCollisionObject* contains the world transform and shape of the object, *btCollisionShape* defines the geometric shape of the collision object such as box, sphere, triangular mesh and convex hull, *btGhostObject* is used for the phenomenon of ghost collision that happens when a surrounding convex hull is used instead of a convex shape, and b*tCollisonWorld* is a container of all collision objects [18].

In the narrow phase, Bullet uses the dispatcher iterates over all colliding pairs given from broad phase, compute contact points of each colliding pair, and stores the information in *btPersistentManifold* structure. Choosing the right collision shape for the simulation is important for which Bullet introduces the following algorithm for choosing a proper collision shape [18].

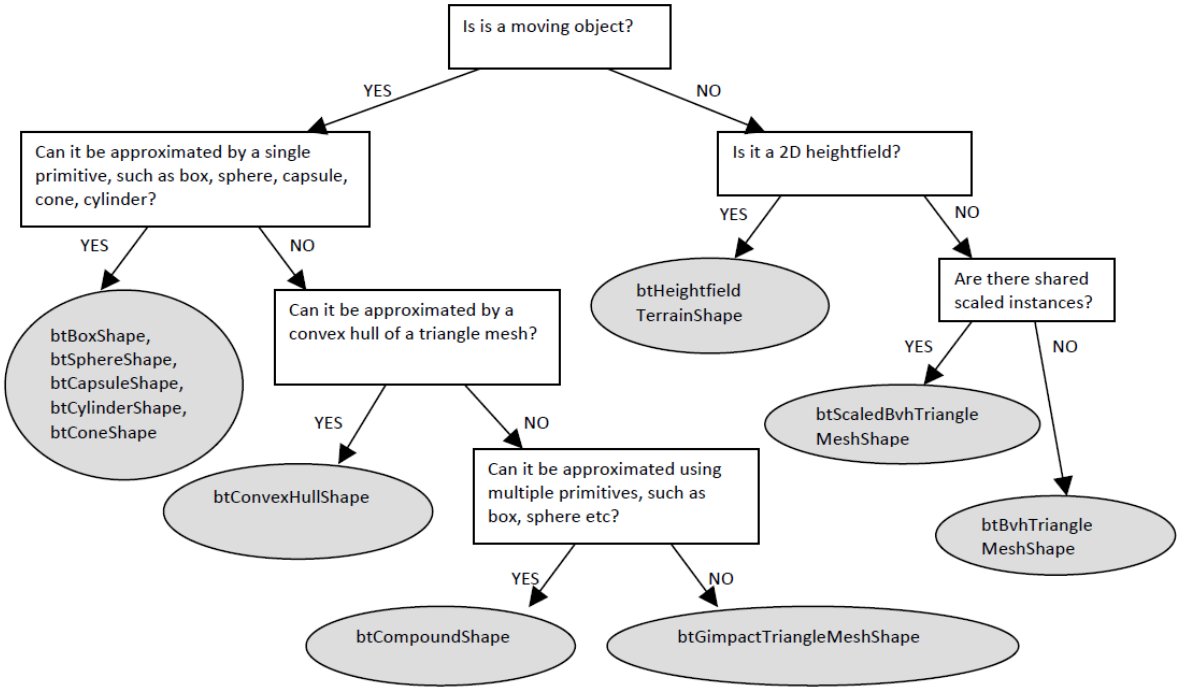

Figure 20. Bullet Physics Manual flowchart for selecting appropriate collision shape [18]

2.4.4. Bullet Rigid Body Dynamics

Bullet uses the class *btRigidBody* to create moving objects and simulate their 6 degrees of freedom (DOF) motion derived from the *btCollisionObject* class. It identifies the geometric and mechanical properties including shape, transform, friction and restitution and adds other features such as mass,

local inertia, velocity, constraints and forces. *btTypedConstraint* class is used for rigid body constraints. The *btDiscreteDynamicsWorld* class includes all of the rigid bodies and constraints and performs the simulation by calling *stepSimulation* function.

There are 3 different type of rigid bodies in Bullet. Dynamic rigid bodies are simply moving or moveable objects. These bodies have constant mass and their motion is updated at each time step of the simulation. Static objects are objects which are not defined to move, while these can still collide with the dynamic objects. Their mass is defined as infinity [18]. Kinematic bodies are objects that can be animated by the user, which can be assigned a infinite mass as well. These can influence on other dynamic rigid bodies' motion but this interaction is one-way coupled (not mutual) as they are not affected by the dynamic rigid bodies' motion.

2.4.5. Motion Update

In Bullet, position and orientation of any rigid body is updated at the center of its mass, i.e., used as the basis for the calculation of its local inertia. Calculation of local inertia depends on the shape of rigid body which comes from the *btCollisionShape* class. The *stepSimulation* function iterates over any rigid body in the world and updates its motion in each timestep. Rendering task in Bullet can be leveraged by MotionState function. Using the MotionStates function saves considerable computational effort because rendering is updated only for moving objects and not for bodies that have not moved significantly in the timestep. Moreover, by means of MotionStates one can detect any possible shifts between center of mass transform and rendered objects. Any interpolated body position is performed by MotionState. Interpolation may be often needed for rendering, however if a non-interpolated position of a body is required, it can be obtained directly by *btCollisionObject::getWorldTransform* or *btRigidBody::getCenterOfMassTransform*. MotionState is mainly used when the body is just created and just enters the simulation. During the performing of simulation, *stepSimulation* updates the body coordinates by *btMotionState::setWorldTrsnsform*.

2.4.6. Creating Dynamic World

Creating a simulation world, applying gravity, motion (transform) initialization, broad phase collision detection that consists of computing AABBs and detecting colliding pairs, narrow phase collision detection that computes contact points and solves collision, and integration or updating motion are all done sequentially in each time step in our dynamic world. The dynamic world is defined in the *btDiscreteDynamicsWorld* class by default, which performs member functions in

stepSimulation to carry out all above-mentioned steps in each timestep. To put in a nutshell, dynamic world is where the simulation is performed. Following lines shows the code of creating a dynamic world in Bullet Physics for running a simulation:

- The objects are first defined for its initialization, which are *collisionConfiguration* for memory management of collision set up, *overlappingPairCache* for broad phase collision detection, *dispatcher* for narrow phase collision detection, and *solver* for solving contacts and calculating impulses which are objects derived from classes *btDefaultCollisonConfiguration*, *btBroadPhaseInterface*, *btCollisionDispatcher* and *btSequentialImpulseConstraintSolver* respectively.

```
collisionConfiguration = new btDefaultCollisonConfiguration();
overlappingPairCache = new btBroadPhaseInterface()
dispatcher = new btCollisionDispatcher();
solver = new btSequentialImpulseConstraintSolver();
```

- Defining a *dynamicsWorld* as an object of class *btDiscreteDynamicsWorld* and initializing:

```
dynamicsWorld = btDiscreteDynamicsWorld (dispatcher, overlappingPair-
Cache, collisionConfiguration, solver);
```

- In case importing a world into Bullet's simulation world is needed, the class *btBulletWorldImporter* can be used:

```
fileLoader = new btBulletWorldImporter ( dynamicsWorld );
fileLoader->loadFile(File Name);
```

  This file is commonly a .bullet or .obj file exported from Blender.

- Gravity is set:

```
dynamicsWorld -> setGravity (btVector3 (0, -9.8, 0));
```

- Creating rigid bodies is done as below. In Bullet rigid bodies are dynamic bodies of class *btRigidBody* which is derived from the *btCollisonObject* class. This means any dynamic body is a collision object and inherits its features such as collision shape, which forms the

shape and dimensions of a rigid body. Each rigid body has its own additional features such as mass and inertia. The added rigid body is initialized with its primary transform after definition is added to the world. Collision shapes include Box, Sphere, Cone, Convex Hull or Triangle Mesh, etc. Material properties such as friction and restitution should also be assigned.

```
btCollisionShape* colshape = new btSphereShape (btScalar(Radius));
CollisionShapes.push_back(colShape);
btTransform  startTransform;
startTransform.setIdentity();
btScalar mass(value);
btVector3 localInertia (0,0,0);
colShape -> calculateLocalInertia (mass, localInertia);
startTransform.setOrigin(x value,y value,z value);
myMotionState = new btDefaultMotionState(startTransforn);
btRigidBody::btRigidBodyConstructionInfo  rbInfo(mass,  myMotionState,
colShape, localInertia);
btRigidBody* body = new btRigidBody(rbInfo);
dynamicsWorld -> addRigidBody (body);
```

- The transform for active objects of the world at each time step is done by *stepSimulation* which performs collision detection and physics simulation.

```
For (i=0, i<Number of desired time steps, i++)
dynamicsWorld ->stepSimulation (time step);
```

2.4.7. Time Integration by stepSimulation

The main input of *stepSimulation* function is the time step. Bullet uses an internal fixed time step of 1/60 (= 0.01666) seconds. When a simulation's time step is smaller than 0.01666 seconds, Bullet will automatically interpolate body movement and put it into MotionStete. The number of simulations to be performed by each *stepSimulation* call is defined by the user and is passed as the second input to the *stepSimulation* function. If the explosion simulation is being performed, updated ex-

plosion force with time is applied each time step. Then two functions named *InternalSingleStep-Simulation* and *synchronizeMotionStates* perform the simulation and motion update. Applied forces are cleared after the end of each time step as below:

```
applyGravity;
applyExplosion();
For (i=0; i<clampedSimulationSteps; i++)
{
InternalSingleStepSimulation(Bullet's fixed time step = 0.01666)
SynchronizeMotionStates();
}
clearForces();[21]
```

InternalSingleStepSimulation is the core of the *stepSimulation* function. In this function which gets Bullet's fixed time step as an input, first unconstrained motion of any dynamic rigid body is calculated by *predictUnconstrainMotion( )*. Collision detection is then performed by means of function *performDiscreteCollisionDetection( )*. The solver resolves the contact problem by *solveConstraints(getSolverInfo)* and after-collision velocities and reciprocal applied impulses are calculated. In the next step, after-collison positions are updated and calculated reciprocal impulses are applied as a central impulse and a torque impulse to colliding pair [21]:

```
integrateTransforms(timeStep);
predictUnconstrainMotion(timeStep);
performDiscreteCollisionDetection();
solveConstraints(getSolverInfo);
integrateTransforms(timeStep);
```

In contact detection procedure, the contact points between two collision shapes are detected. These contact points are preserved in an array called manifold, thus the manifold size is equal to number of contact points. The function iterates over contact points in the manifold and calculates total reciprocal impulse of the colliding pair by means of the calculated reciprocal impulse at any contact point by *btSequentialImpulseConstraintSolver* [18].

2.5. Preprocessing

The modeling, simulation, and visualization are three major steps of using the software package and herein referred to as preprocessing, analysis, and postprocessing. Modeling the geometry of the structure, assigning mechanical modeling properties, discretization based on Voronoi algorithm and creating constraints among fractured rigid bodies are called preprocessing stages and are performed in Blender. The explosion implementation and contact force retrieval cannot be performed by the existing code which are the main features introduced through Bridge Demolition Add-on (BDA) into Blender. The force by explosive blast wave is formulated based on explosion theories that are coded in Python and added to Blender. Moreover, the contact forces retrieval is not available in existing Blender and is also coded in Python as an add-on feature. BDA provides the user interface dialog box that enables user to enter modeling input parameters for explosion demolition simulation, and the post-processing panel for contact force, predicted explosion loading, estimated debris generation and force/stress contour visualization as the outputs. There are some required steps that should be done in Blender for modeling before using BDA, which referred to as preprocessing. This stage includes geometry modeling, discretization, building constraints, sequence modeling, and finally set up rendering options.

2.5.1. Geometry Modeling

The geometry modeling in this project is performed by leveraging Blender, which is a free and open-source 3D computer graphics software toolset. Blender is widely used for creating animated films, interactive 3D applications and even video games. Moreover, Blender integrates Bullet as one of the main physics engines and shares some common modeling procedures, e.g., Voronoi tessellation for discretization. In this section, some basic modeling functions of Blender are introduced. Interested readers are referred to Blender User Manual [22] for further reading.

The first step models the major structural elements such as piers, pier cap and deck. A homogenized element may be modeled with equivalent mass and stiffness as far as a monolithic behavior is assumed. After all structural parts are separately modeled, they are then merged with rigid connections. Figure 21 shows a 2-span bridge with span length of 13 meters and total mass of 196 tons in Blender.

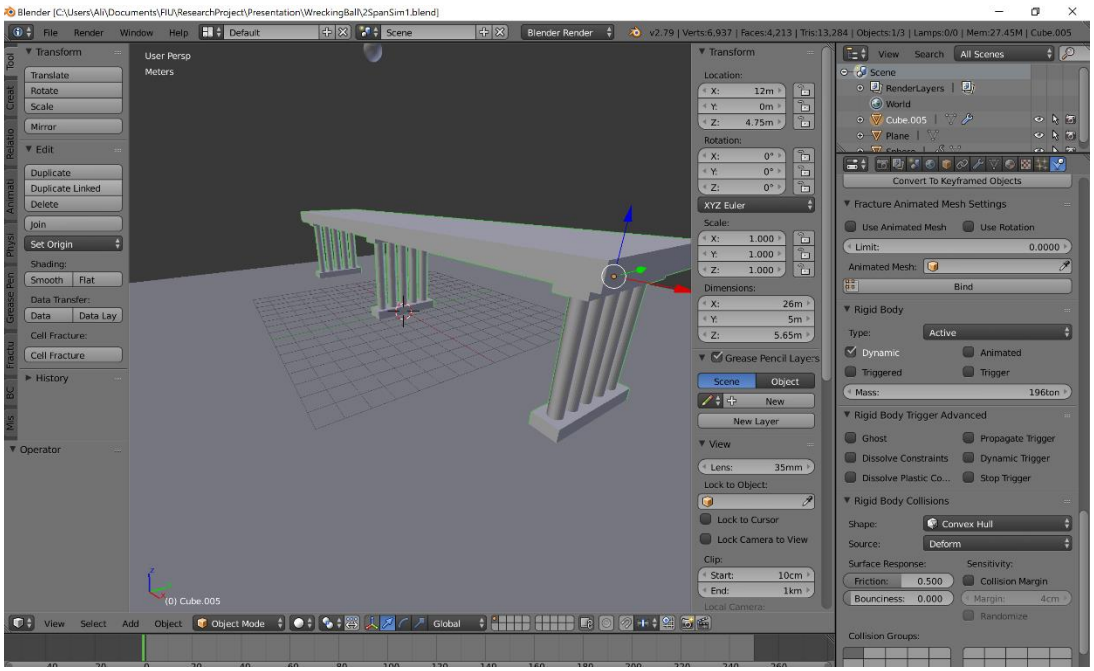

Figure 21. Modeling a 2-span bridge in Blender

The primitive shapes such as cube, sphere, cylinder and other polyhedrons are available for modeling in Blender, meaning each shape is composed of polygonal faces, edges and vertices. Sphere is not a perfectly smooth curve, but piecewise linear. These shapes can be altered with Mesh Modeling option whereby users can move the vertices and extrude faces, etc. to change the shape and dimensions of desired object [22].

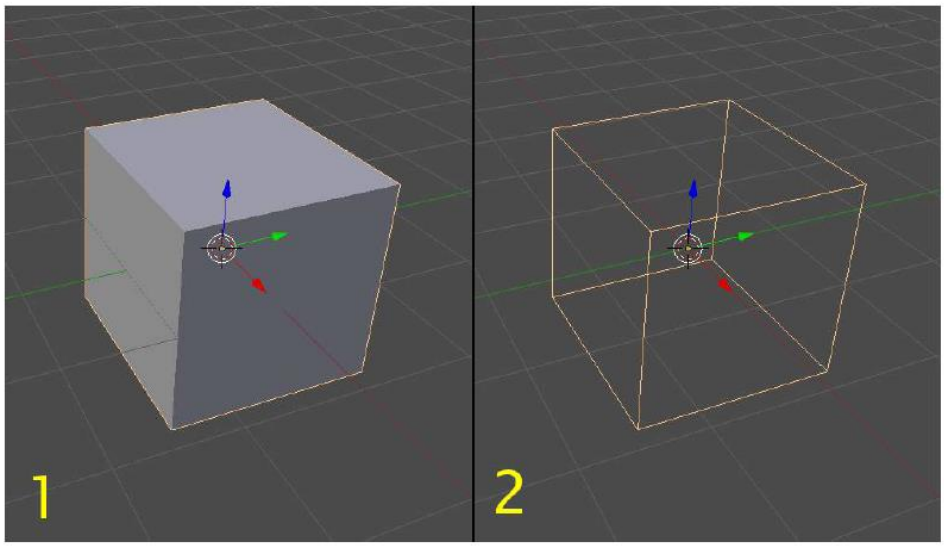

Figure 22. Cube, a primitive mesh shown in 1) solid shading and 2) wireframe shading [22]

Mesh is a principal model object in Blender. Each object has an origin point which defines its location in 3D space [22]. Location, dimensions, scaling factors and other transform properties can be set in Transform tab.

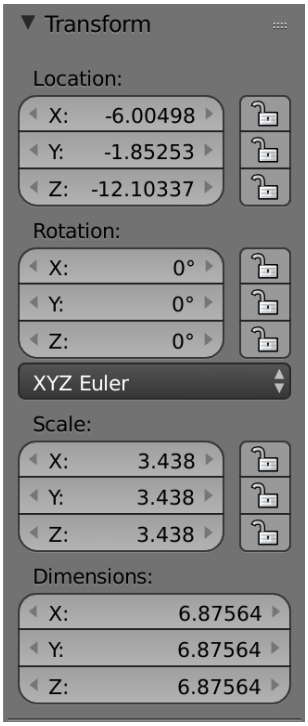

Figure 23. Transform tab

Selecting an object can be done in two modes: Object mode and Edit mode. Each mode enables different tools and options to edit any object:

In the Object mode, adding a mesh to the world is done by hotkey Shift-A. Any type of object such as Mesh, Curve, Surface, Metaball (e.g. capsule, ellipsoid, etc.), text and others can be added to the world. Adding a force field or a lamp or a camera is also possible. Duplication of an object and regeneration of identical copies of an object can be performed by hotkey Shift-D. Hotkeys Shift-R and Shift-S can be used to rotate and scale an object respectively. Grabbing and moving object along a desired axis can be done with hotkey Shift-G-X (or Y, Z depending on the axis). Merging option can be used to create a compound object. By selecting all objects to be merged, the merging procedure can be completed by hotkey Ctrl-J [22].

There is an important consideration needs to be made when merging objects. In any merging action, it will be likely that some vertices or faces may overlap or intersect (Figure 24), which may

cause technical issues when using along with the Blender fracture modifier [23] for use in model discretization. It is recommended to use Boolean modifier (Figure 25). The Boolean modifier can apply operations such as intersection, union and difference. By using these set of operations, extra faces and vertices are simply extracted from the merged compound object (Figure 26).

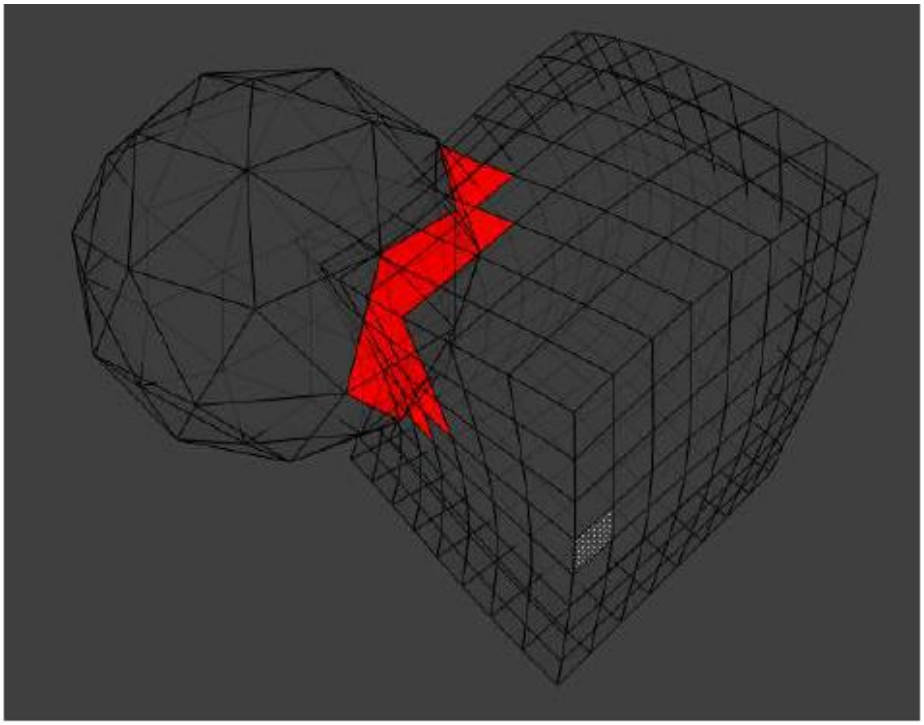

Figure 24. Intersecting faces [22]

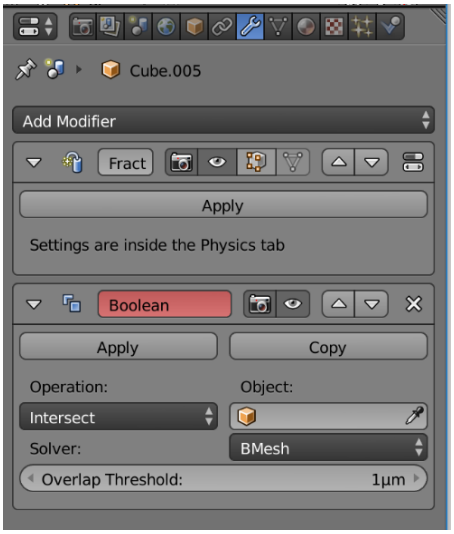

Figure 25. Boolean modifier to extract overlapping vertices, edges and faces in a compound object

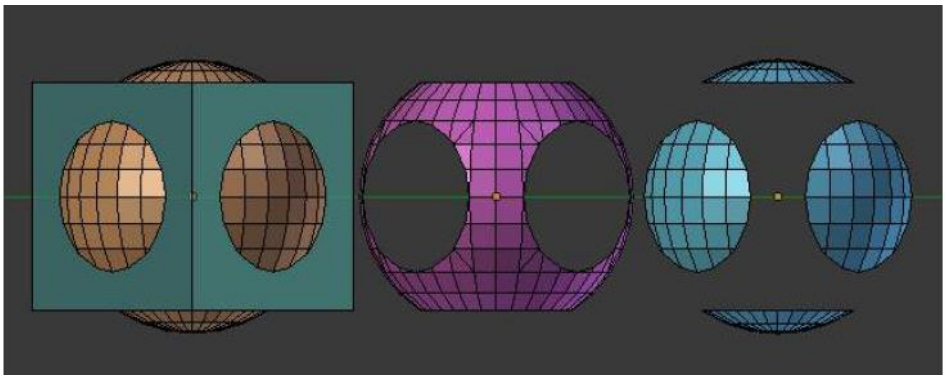

Figure 26. Difference, Union and Intersect in Boolean Modifier to create a compound shape [22]

Deleting an object can be easily done by hotkey X or simply hitting Delete key. It is required to switch to Edit mode (Figure 27) to make a detailed change of the geometry by directly selecting and applying the changes to vertices, edges and faces in the object (Figure 28, 29).

Figure 27. Edit mode select mode buttons [22]

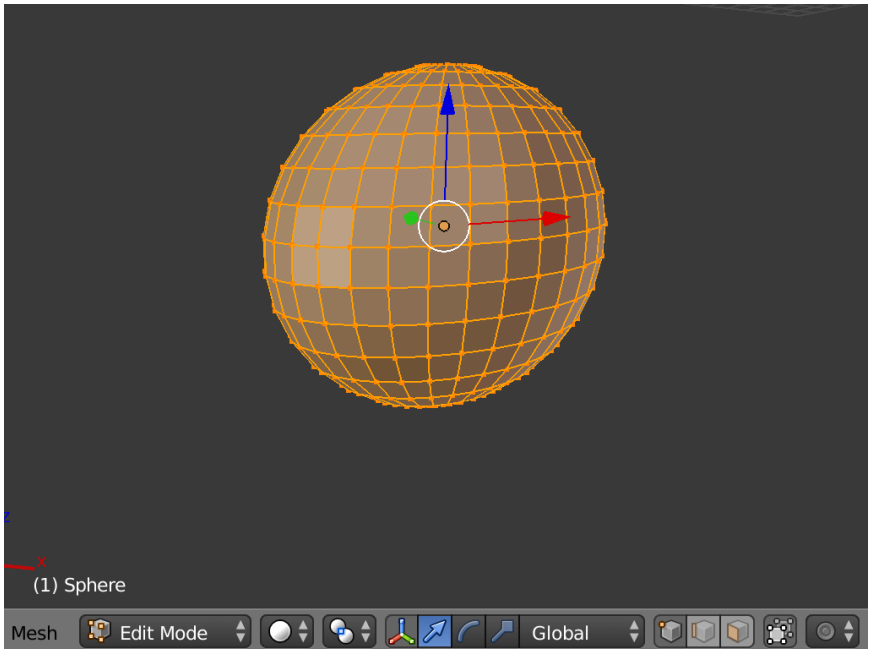

Figure 28. A sphere in Edit mode with all of vertices selected. Edges and faces can be selected separately

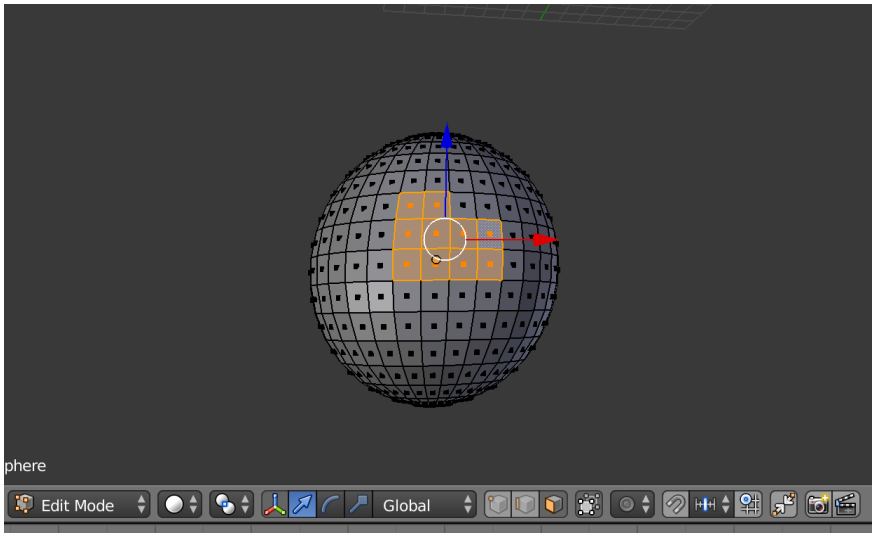

Figure 29. A sphere in Edit mode with some faces selected

Any geometric operation implemented in Blender can be applied to vertices, edges and faces such as extruding, intruding, scaling, rotating, mirroring to create modified geometries (Figure 30).

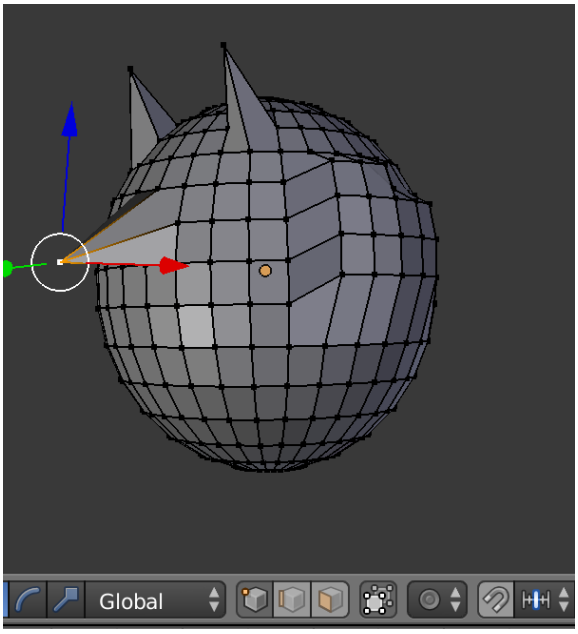

Figure 30. A modified geometry by operations in Edit mode

Modifiers in Blender are the tools which can be utilized to alter the way objects are displayed or rendered (Figure 31). There are four categories of modifiers: *Modify*, *Generate*, *Deform* and *Simulate*:

Modify modifiers just affect the data of an object and do not directly influence the shape of an object.

Generate modifiers change the appearance of an object and add new geometry.

Deform modifiers change the shape of an object but do not add new geometry.

Simulate modifiers can add visual effects and simulations when a particle system or a physics system is defined and enabled. It can create an animated realistic ocean or generate simulation and visual effects like collision, exploding or smoke.

Figure 31. Modifiers menu [1]

Any object modeled in Blender as a rigid body can be active or passive. An active object can be a dynamic rigid body or animated (user-controlled) rigid body. A passive rigid body is initially at rest who does not move until it reacts to a collision. Parameters related to dynamics of a rigid body can be set in Rigid Body Modifier tab as shown in Figure 32.

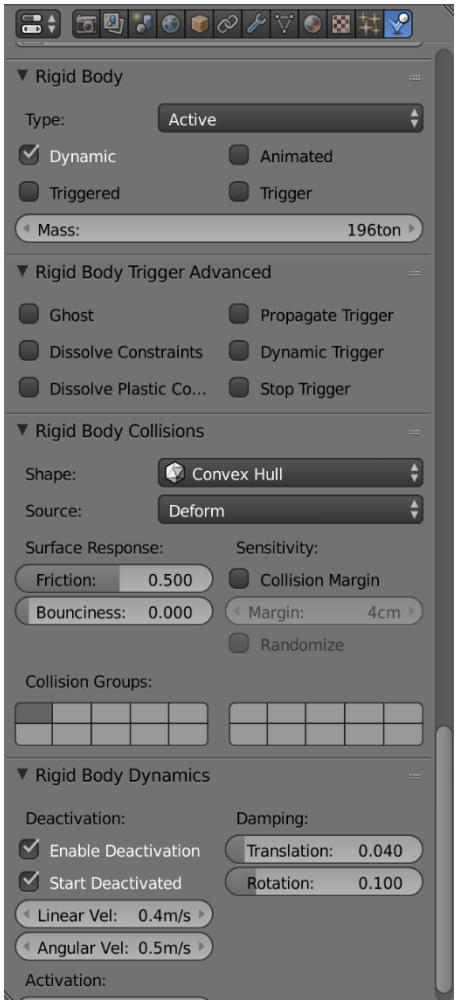

Figure 32. Rigid body parameters and options in Rigid Body modifier

Physics modeling parameters such as mass, friction coefficient, and bounciness (coefficient of restitution or contact damping) can be set. There are options called Enable Deactivation and Start Deactivated that play an important role in the demolition simulations. If these options are not checked, the pre-fractured bridge may start to collapse from the very beginning of simulation, even before the wrecking ball has not touched the structure. This behavior also strongly depends on the constraints defined for the shards. Tuning of these options will be further discussed in the later sections.

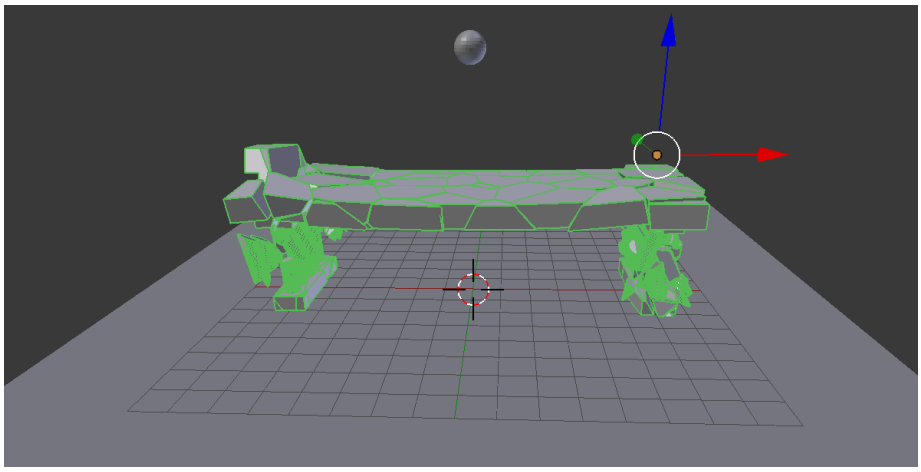

Figure 33. Collapse of the bridge model with pre-fracturing and not selecting start deactivated option

2.5.2. Discretization and Fracturing

There are two methods available in Blender to implement the discretization in the developed bridge model: (a) Cell Fracture add-on and (b) Fracture Modifier which is a more recent development compared to the former. We will briefly introduce the (a) Cell Fracture add-on first as below.

The Cell Fracture add-on is not a default option in Blender, which therefore should be imported first. The import procedure is shown in Figure 34, which can be done by checking Object: Cell Fracture in the Add-ons tab as below. Then Cell Fracture add-on is added to the Tools bar as shown in Figure 35.

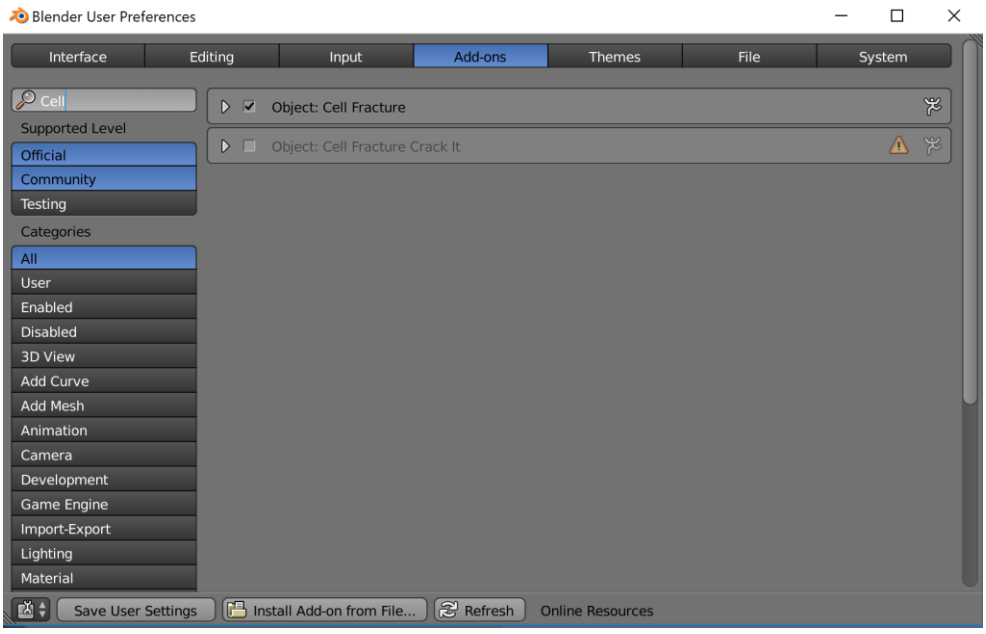

Figure 34. Enabling cell fracture add-on in Blender's user preference.

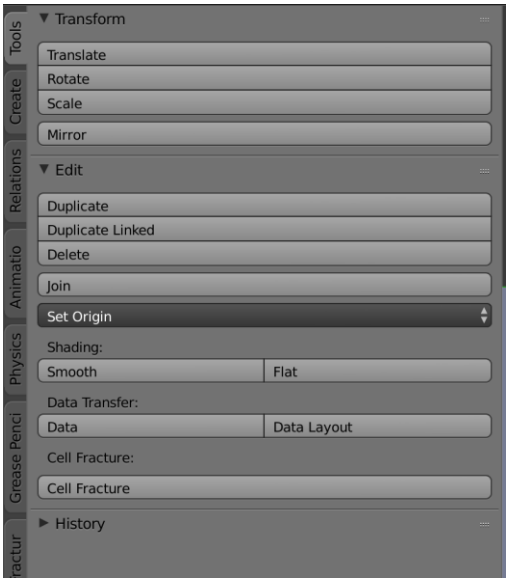

Figure 35. Cell Fracture add-on enabled in the Tools bar.

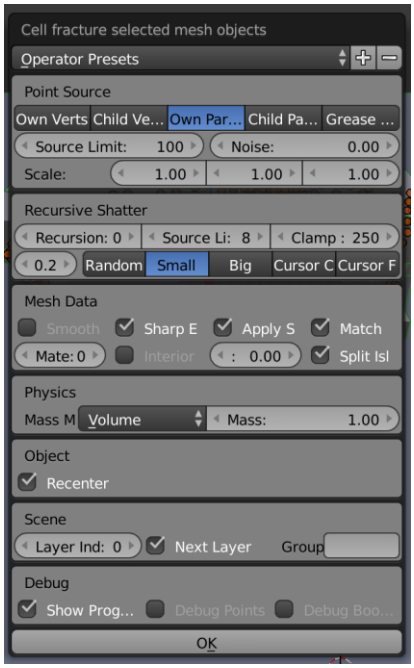

Figure 36. Cell Fracture features and parameters.

While bridge object being selected, clicking the Cell Fracture opens the Cell Fracture window as shown in Figure 36. The Source Limit parameter in the add-on determines the maximum number of shards that can be generated. Noise value is a value between 0 and 1, which accounts for irregularity of fracture sizes. The value 0 indicates completely uniform shards in terms of shape and size, while 1 indicates the maximum nonuniformity. The Recursion parameter helps to generate

smaller shards. The shards' mass can be set uniform or proportional to their volume by controlling the Mass parameter. By checking the Recenter option, the physics properties of each shard are calculated based on its own origin. Checking the Next layer option makes the discretization to be shown in the second layer rather than the main layer of the scene. By pressing OK, the bridge is discretized. An example is shown in Figure 37.

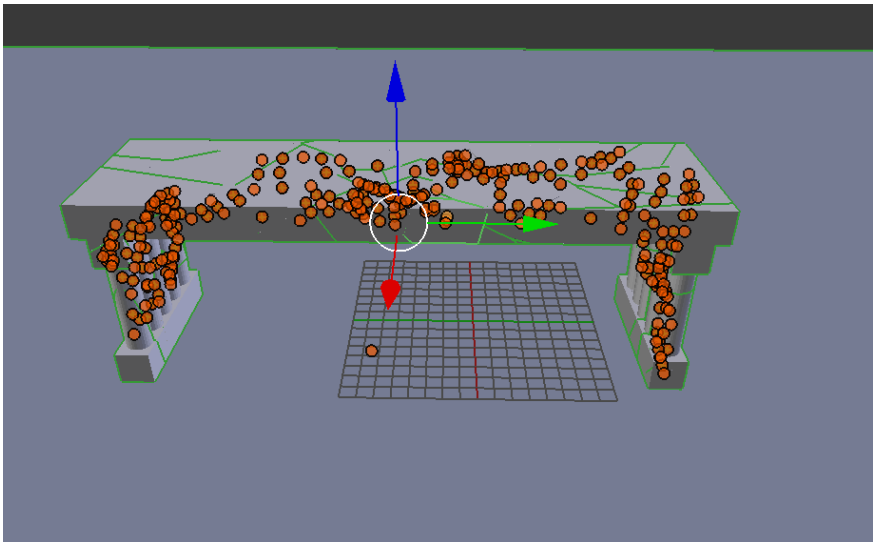

Figure 37. Discretized bridge by using cell fracture add-on

Now, rigid body properties will be assigned to new rigid bodies generated by fracturing, which can be done by using Physics Body Tools bar shown in Figure 38. Selected the shards can be set as active objects with the Add Active option. The volume proportional mass can be calculated with Calculate Mass option and then Concrete option can be selected as the designated material.

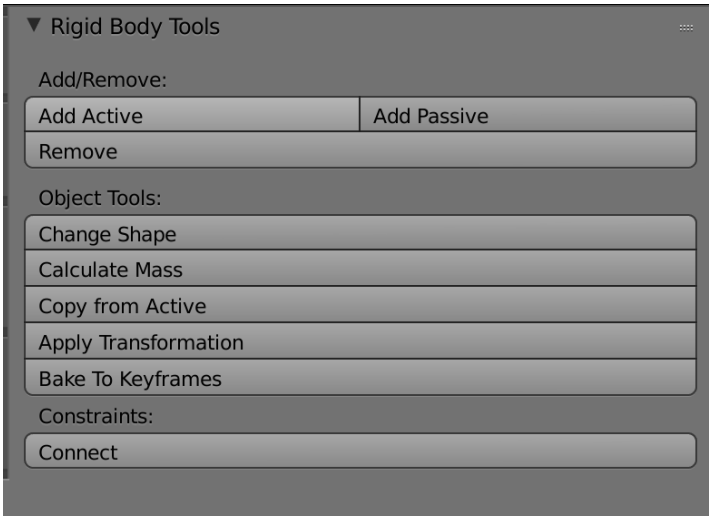

Figure 38. Physic Body Tools bar for assigning rigid body properties

In the recently developed (b) Fracture Modifier, a set of fracture algorithms are available. In this study, Voronoi Boolean algorithm is selected for the model pre-fracturing. There are also some options available for solver from which Carve is selected. The number of shards can be chosen to determine how the bridge model is to be fractured, which can be selected depending on the computational cost and simulation fidelity. A large number of bodies will be more realistic, but is computationally much expensive, as it requires to simulate more number of broken pieces and their interactions in every time step.

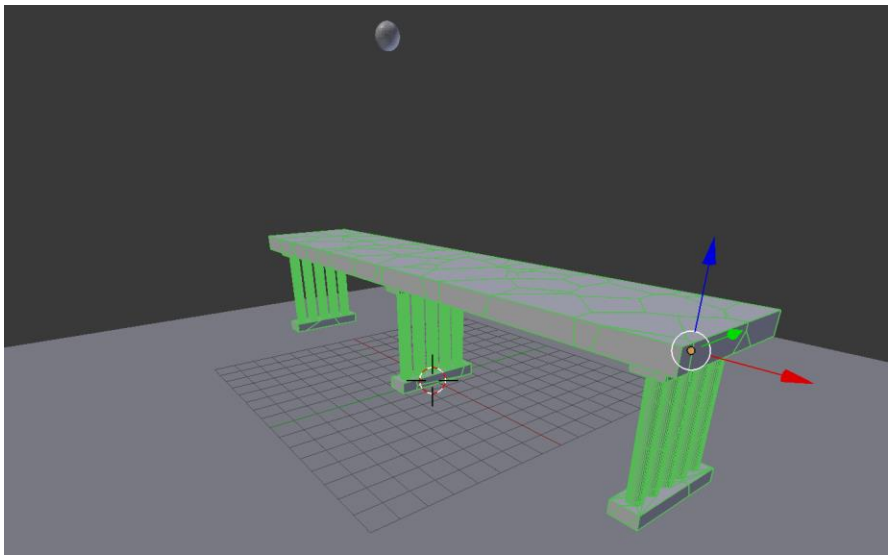

Figure 39. Discretization of a 2-span bridge model by Fracture modifier

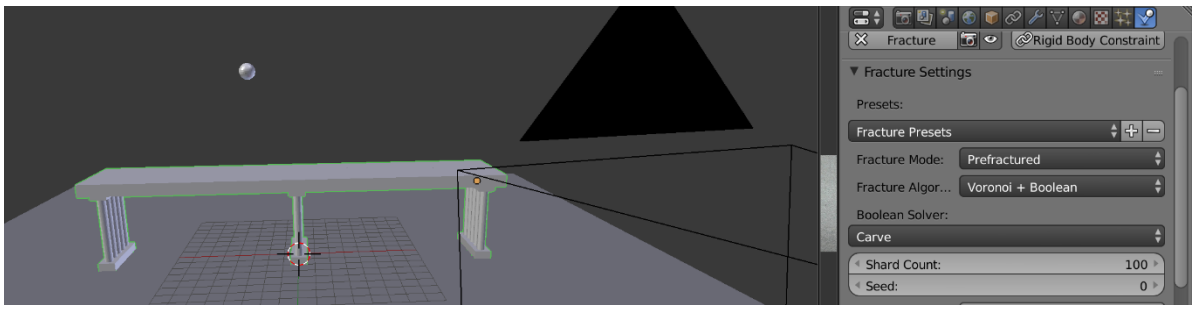

Figure 40. Fracture settings

The constraints between the shards can be set in the Fracture Constraint Settings tab (Figure 41). Here, one can enable to use constraints, set them breakable, and select the type of the constraints between the shards. The example in Figure 41 shows Hinge type is selected as the Constraint Type and Vertex is chosen as the Constraint Method. Number of constraints per mesh island can be also set. The constraint threshold to be broken can be set in Constraint Breaking Settings (Figure 42).

Figure 41. Fracture Constraint settings

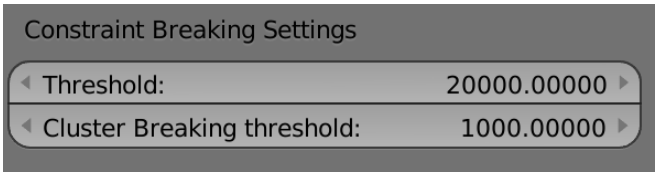

Figure 42. Constraint breaking settings

2.5.3. Building Constraints

With the pre-fracturing performed, the bridge model is geometrically discretized into hundreds of separate rigid bodies, then the constraints between these objects need to be set up. To this end, Bullet Viewport Constraints Tool add-on [6] is used. The add-on is not a default option in Blender, which therefore should be imported first as well (Figure 43). Once installed, the option is added on the left tool bar in Blender's 3D view as shown in Figure 44.

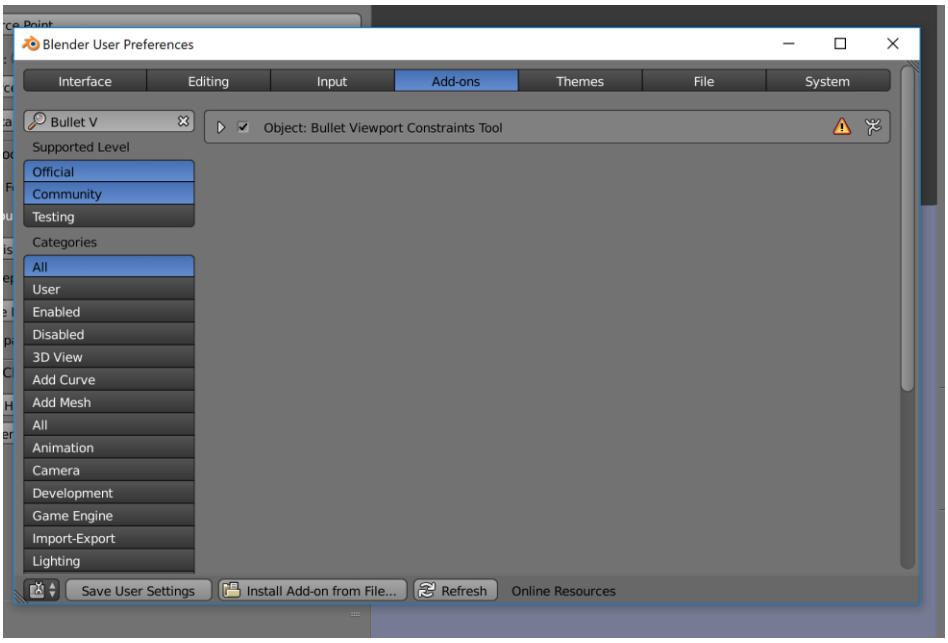

Figure 43. Adding Bullet Viewport Constraints Tool

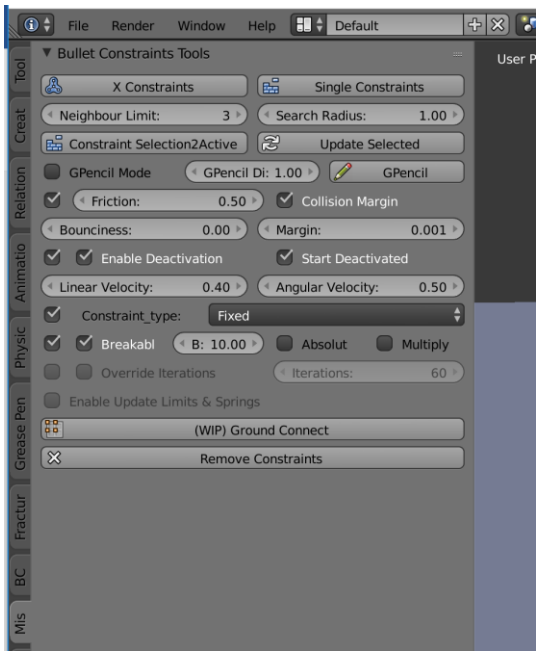

Figure 44. Bullet Viewport Constraints Tool user interface

The corresponding shards need to be selected to assign the building constraints in the discretized bodies. The same constraints may be applied to all the shards in the bridge model, or different constraints may be selectively assigned to different parts of the bridge. The Neighbor Limit value in the Bullet Constraint Tools window (Figure 44) determines the maximum number of surrounding shards to be considered for building a constraint around the target shard, i.e., for nonlocal interaction of rigid bodies. By checking Friction, a frictional contact is considered between the shards, for which the friction coefficient (such as Coulomb friction) and the restitution values need to be set. The Enable Deactivation option enables the discretized model is dynamically deactivated if the external forces applied at the interface is not high enough compared to the constraint threshold, i.e., not strong enough to break the constraint. This option helps keep the computational cost more manageable by lowering the number of the necessary contact detection that is computationally expensive in general. Linear Velocity and Angular Velocity are the constraint thresholds for activation to translation and rotation respectively. Constraint Type option enables to choose desired type of constraints including fixed, hinge, slider, point, etc. Pressing X Constraints button at the top shows the built constraints on the top right corner of Blender graphical user interface. An example is shown in Figure 45.

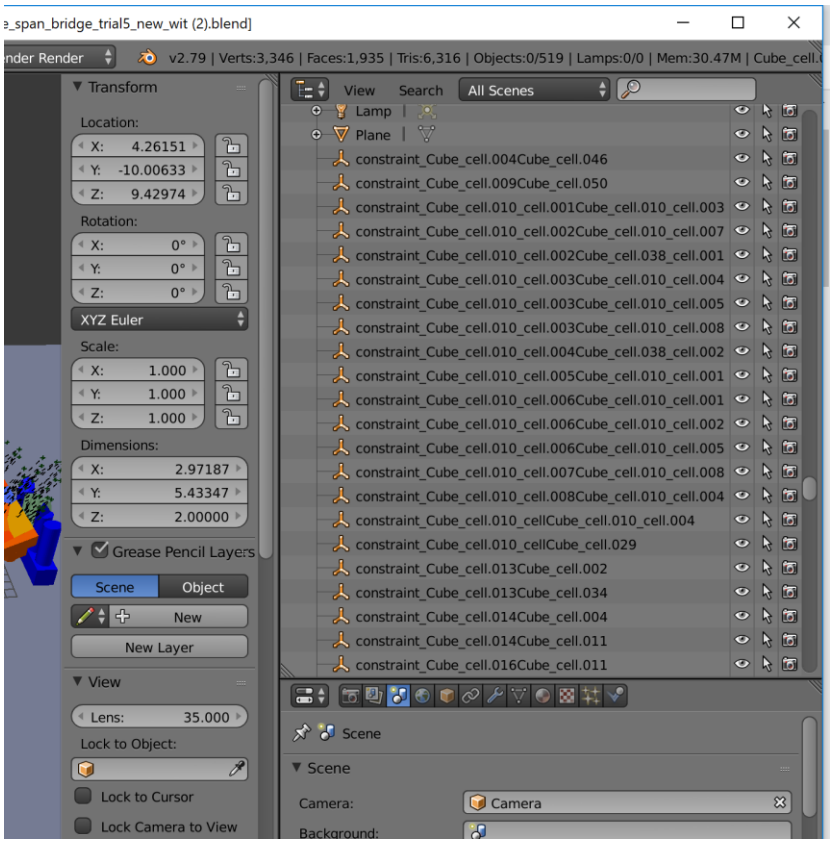

Figure 45. Constraints created

2.5.4. Sequence Modeling

There are numerous cases that movement of an object needs to be manually animated in a simulation. For example, a wrecking ball is pulled up by a cable connected to a crane, and then dropped by gravity to impact the bridge structure. Then it's pulled up again for the next hit. This sequentially controlled movement can be modeled in Blender. The object for animation should be selected and checked as Animated in Dynamics tab (Figure 46). By using Dope Sheet tab and setting Keyframes in a series of desired frames, one can change the dynamics of the wrecking ball from animated to dynamic motion or vice versa (Figure 47).

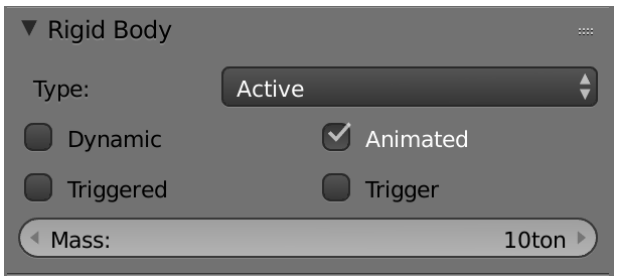

Figure 46. Animated option for considering animated movement of a rigid body in a simulation

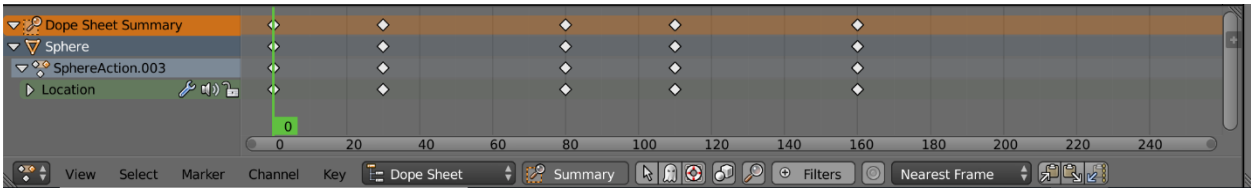

Figure 47. Blender Dope Sheet, setting keyframes for a specific action

2.5.5. Rendering

Once the simulation of bridge demolition is completed, the results needs to be visualized for which sophisticated rendering is needed. One can monitor how the rendered image of an object appears by controlling the four factors: (i) camera, (ii) lighting of the scene, (iii) object's material and (iv) general render settings. Blender adopts two different render engines: Blender Renderer and Cycles. Regardless what render engine is used, the rendering commonly is performed in the following steps: First, the location of the (i) camera is defined. The (ii) lighting of the scene is then determined. Instead of using conventional lighting options, emission planes can be used to provide enough light to the scene. This is done by adding some number of plane meshes in the scene with relatively greater dimensions compared to the developed bridge model, and selecting their positions and orientations in a way that they can emit sufficient amount of light to the scene (Figure 48). Then one should go to Material modifier and define a new material and then select Emission and set the strength of the emission in the Surface tab.

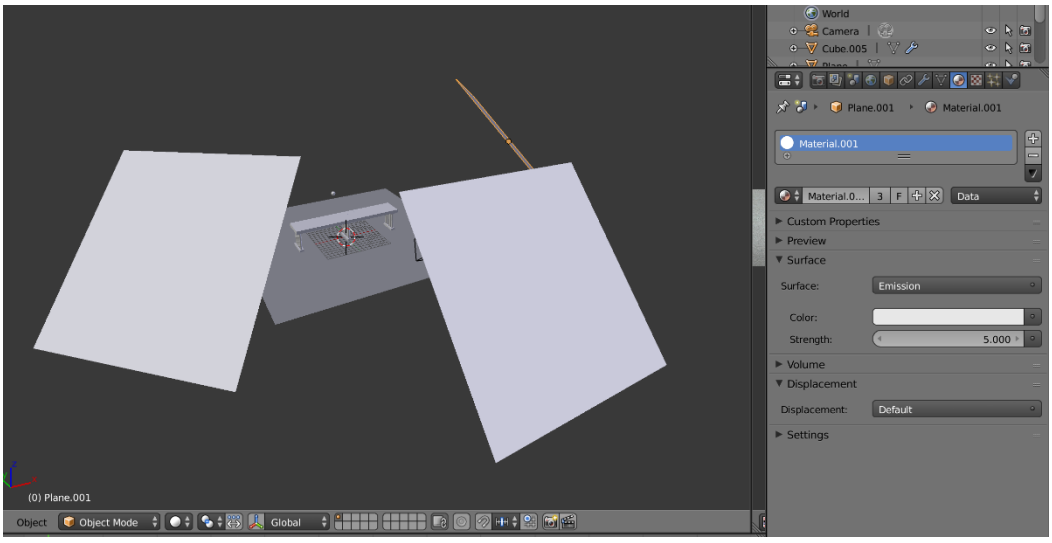

Figure 48. Providing light to the rendering scene by big emission planes

The (iii) materials then can be assigned to the objects. It's also possible to assign a texture, which is optional. Material in Blender can be defined by clicking the Materials button (Figure 49). To add a new material, click + in the material box. A material has a wide range of properties such as preview, diffuse shaders, color ramps, shading, transparency, etc. which can affect the final rendering.

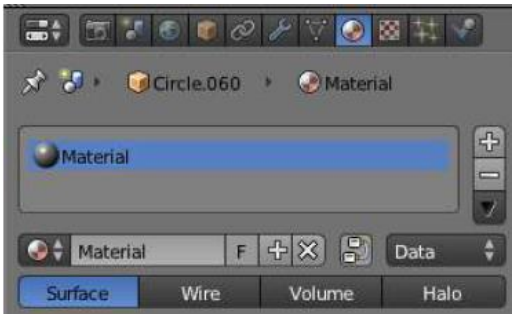

Figure 49. Materials Panel

User may switch to Cycles Render mode and set the (iv) general render settings like start and end frame, frame rate, processor device, address to save the video, etc. in Render settings tab (Figure 50).

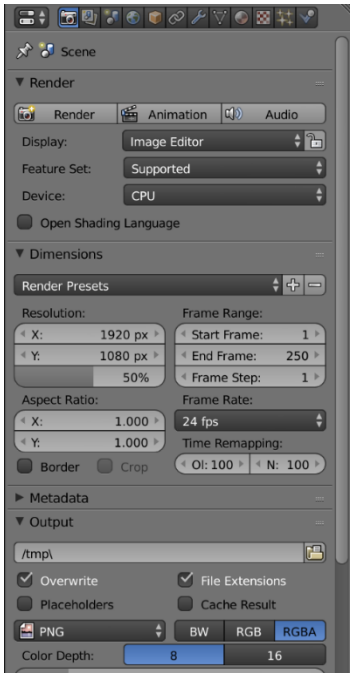

Figure 50. Rendering settings

2.6. Simulation by BDA and Postprocessing

After finishing preprocessing and getting the add-on installed and added to Blender, the bridge model is prepared for a demolition simulation by BDA. If the desired method of demolition is mechanical method using a wrecking ball, leave the Explosion option unchecked in Explosion panel as discussed above and shown in Figure 51. User may click Show Total Mass of Particles button as shown in Figure 52 to compute the exact mass of the discretized bridge model.

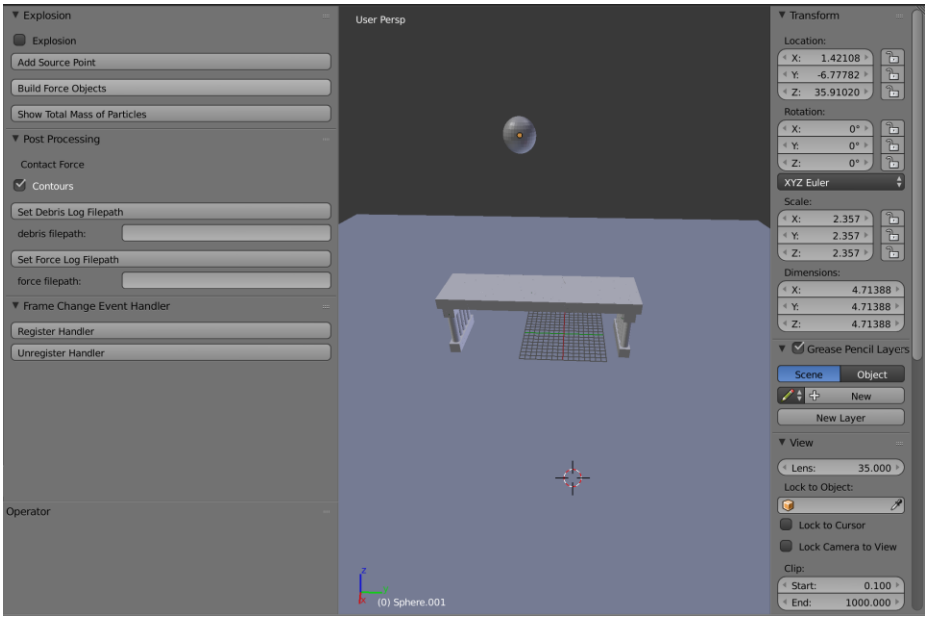

Figure 51. Leaving Explosion unchecked in Explosion panel for wrecking ball demolition simulation

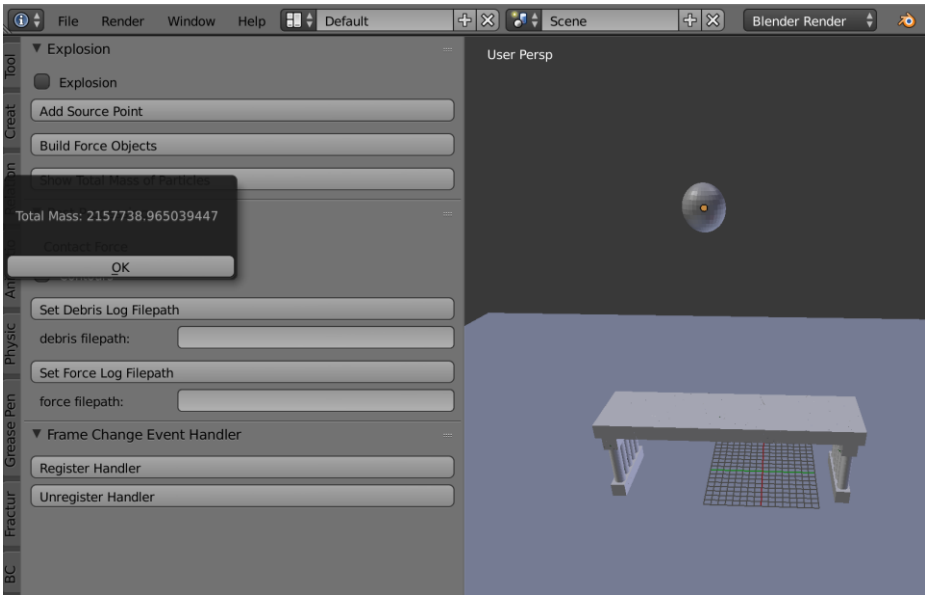

Figure 52. Computed mass of the developed bridge model

Contact force contour visualization is checked by default, which however may be unchecked to disable the contour visualization during simulation. Set the file path by pressing Set Debris Log Filepath to get the debris propagation data. This will write the locations of rigid bodies in the first and the last frame to a text file whose path is selected by the user. This will allow for estimating the maximum distance traveled by each fractured body. Set the file path by pressing Set Force Log Filepath to get the contact forces calculated during the simulation for each frame. This will write the calculated contact forces each timestep of the simulation to a file whose path is selected by the user. Figure 53 shows an example where the paths are set for the output files in the user interface.

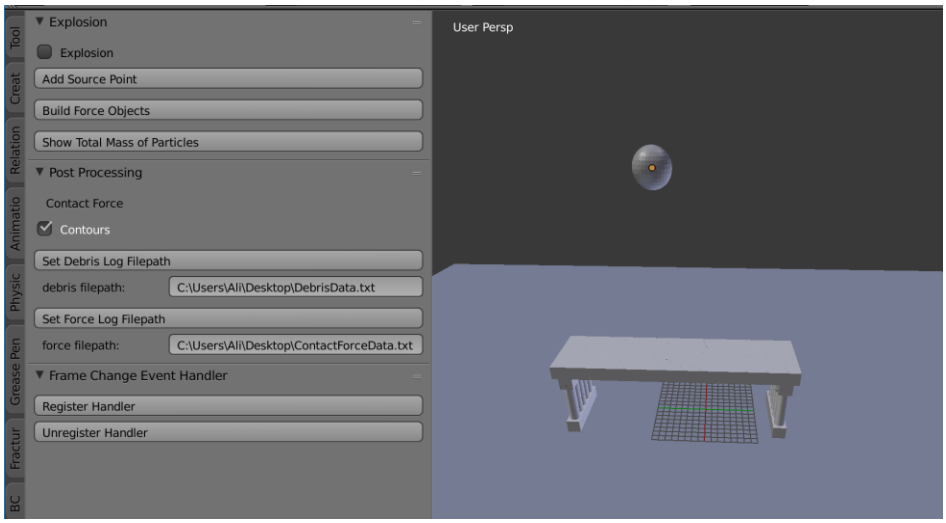

Figure 53. Paths set for the output files in the interface

The model's constraints can be hidden by opening the Python console in Blender and enter command lines as shown in Figure 54. Event handlers can be automatically created by the program by activating Register Handler in the last panel. Then play the simulation.

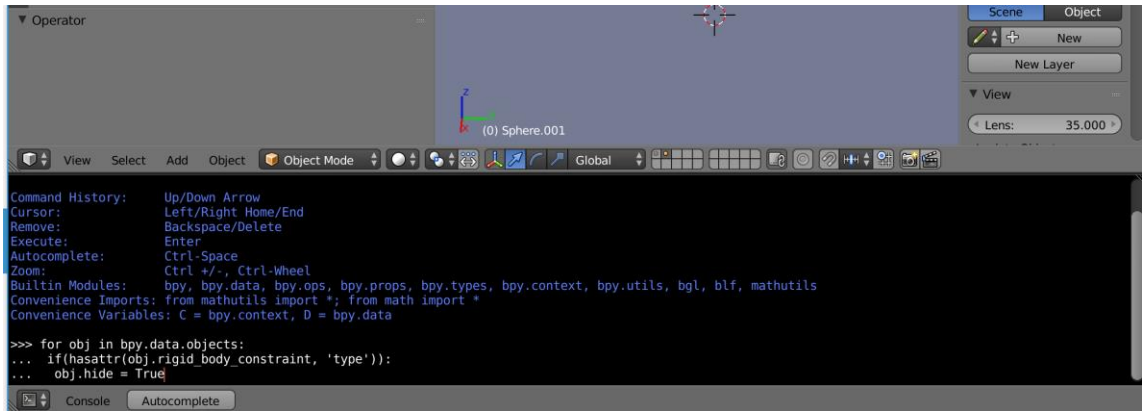

Figure 54. Hiding constraints

At the beginning of simulation, the contour will be shown all in dark blue (Figure 55) showing the minimum contact force values. Figure 56 shows contact force contours as a wrecking ball hits the bridge deck. The maximum contact force values are indicated in darker red.

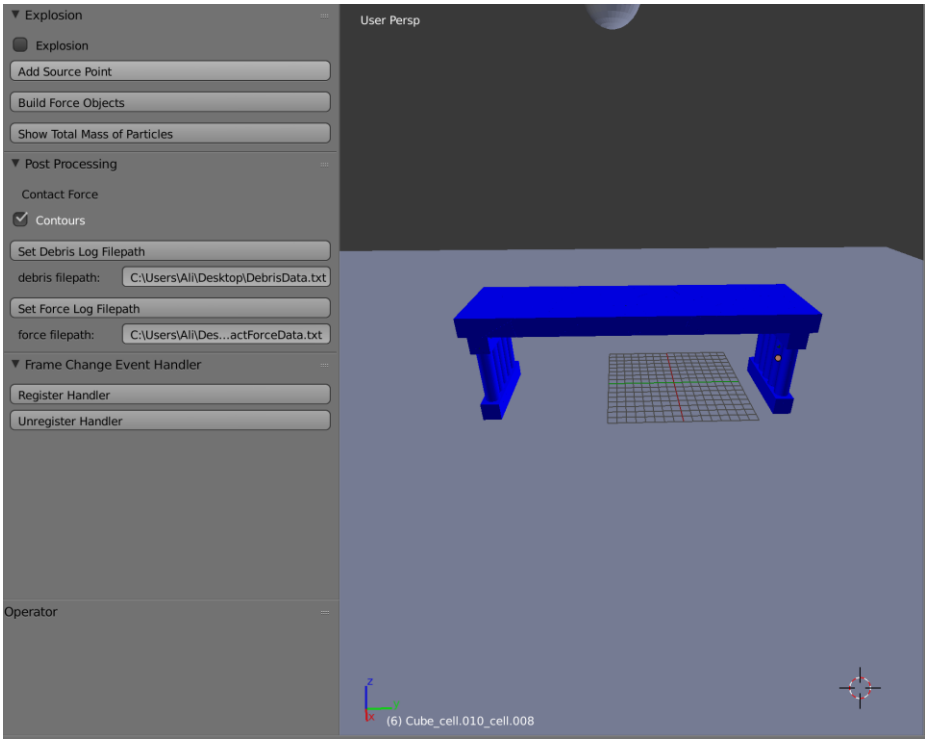

Figure 55. Minimum contact force values

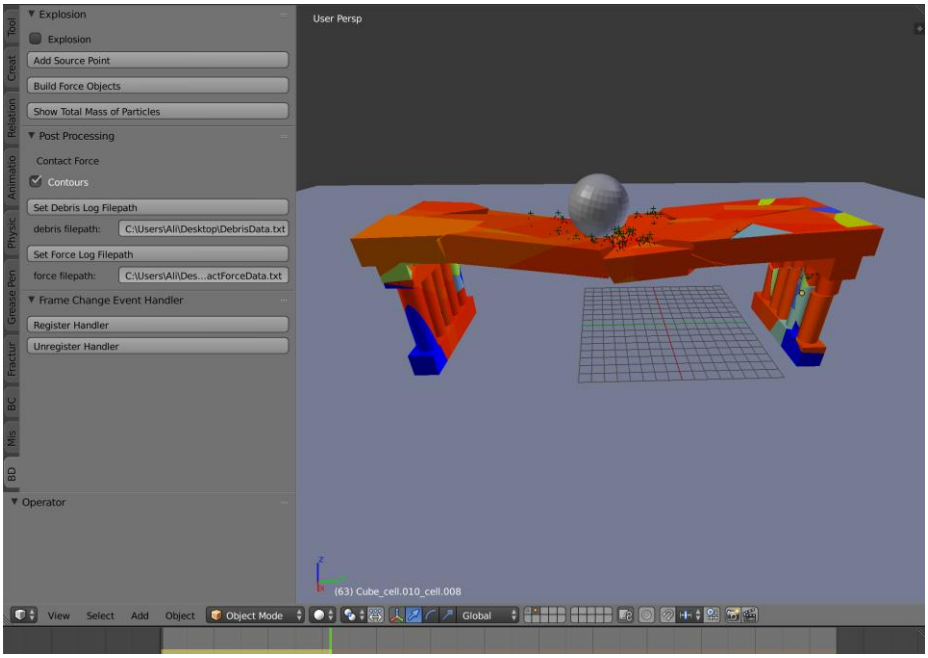

Figure 56. Contact force contours as wrecking ball hits the deck

For the demolition by explosion, the Explosion option should be checked in the panel. With the Add Source Point option, (unlimited number of) new explosion source points can be defined. As shown in Figure 57, by clicking Add Source Point, a new row is generated and the user can define an explosion source point location by entering X, Y and Z coordinates and the amounts of explosives related to each source point. After defining the desired explosion source points, Build Force Objects can be clicked. This will create empty objects which are used for applying the explosion force in three dimensions.

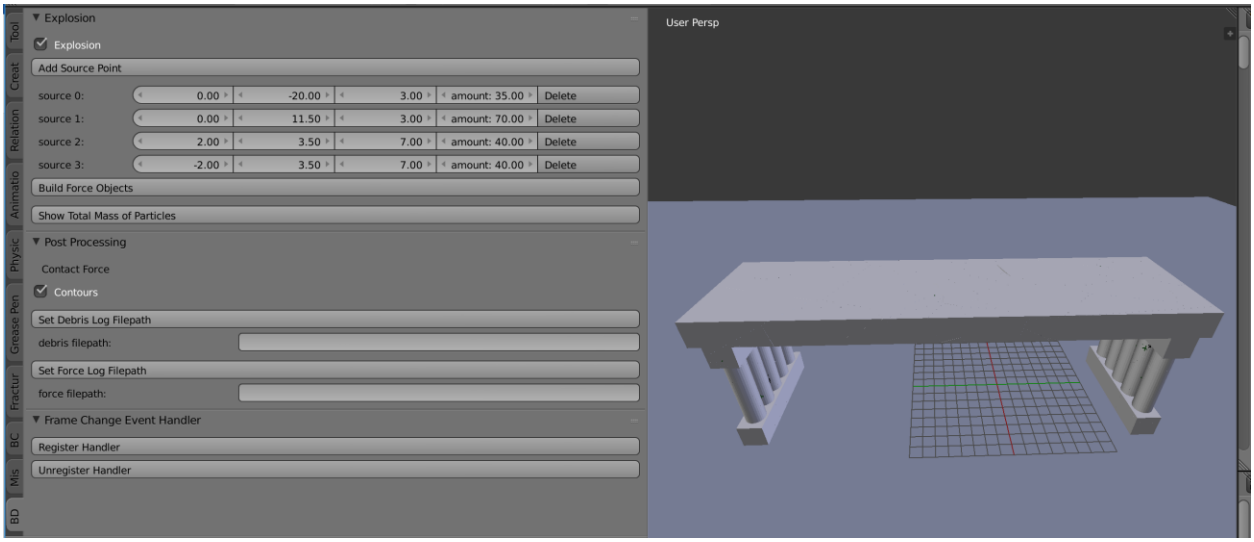

Figure 57. Defining explosion simulation parameters

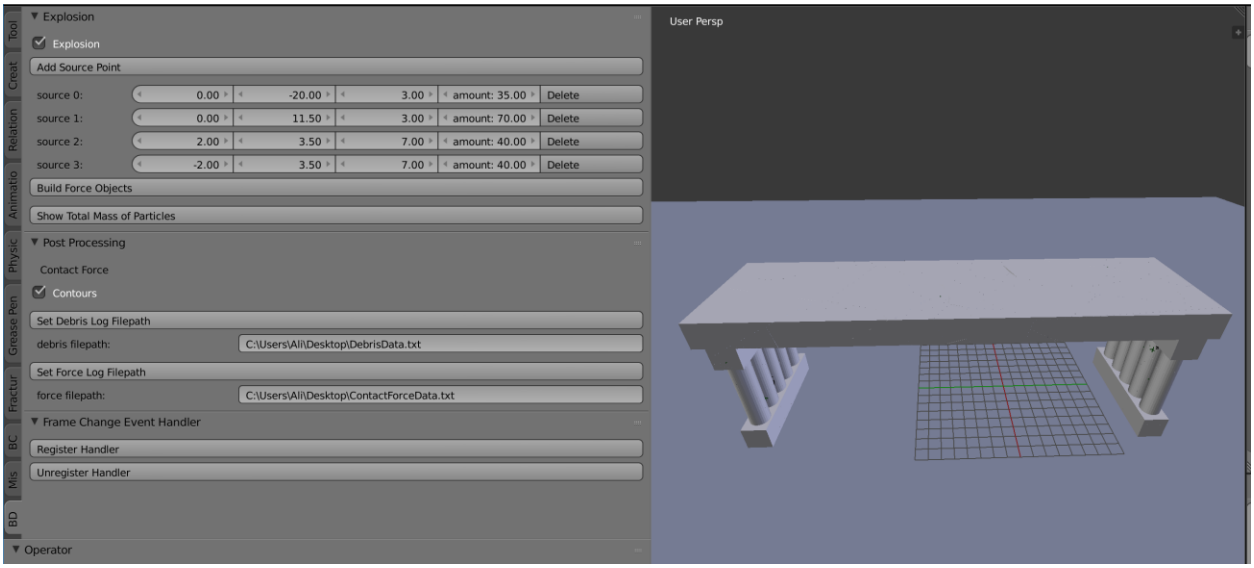

Figure 58. Paths set for the output files in the interface

Paths and filenames need to be set for debris propagation data and computed contact force as simulation output using Set Debris Log Filepath and Set Force Log Filepath (Figure 58). With Explosion option checked, an extra text file will be created in the same path which contains the magnitude of explosive force applied, the projected area and the normal vector of each rigid body in each frame from beginning till the end of simulation. With Register Handler activated in the last panel, event handlers are automatically created by the program. Figure 59 shows a simulation example of demolition by explosives.

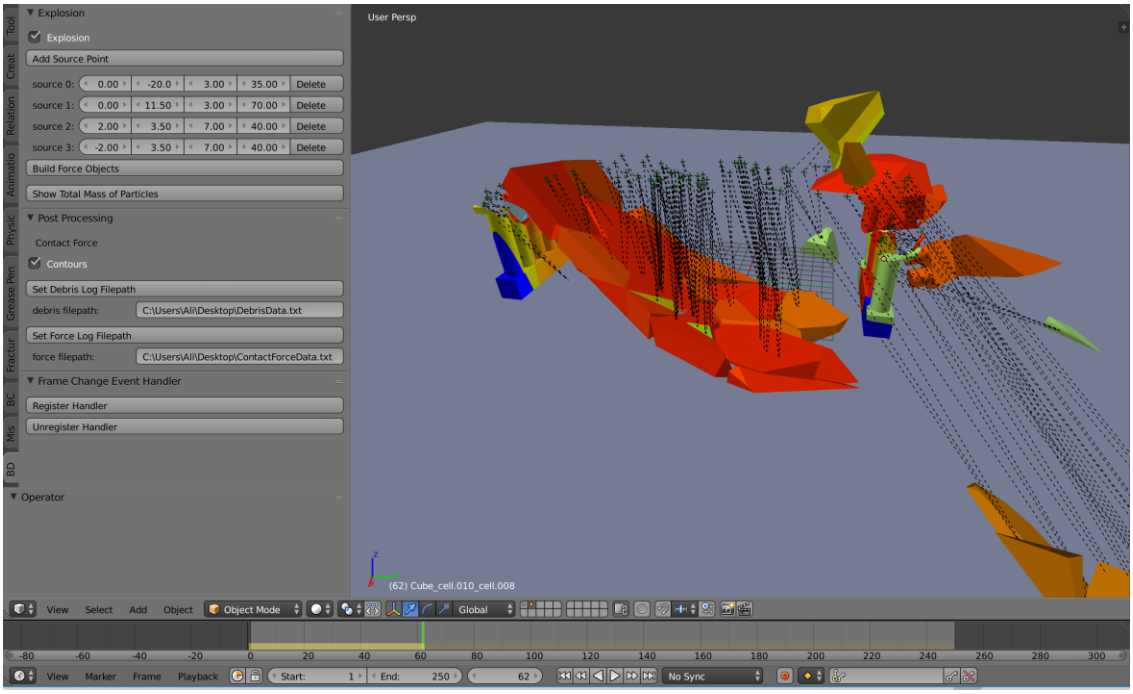

Figure 59. Explosion simulation example performed by BDA

2.7. Calibration, Verification and Examples

The objective of this task is to verify the numerical issues in the developed program. Any major programming issues and errors will be resolved and debugged through the verification process. The program will be then validated using available data and video recordings obtained from the previous demolition projects. The fidelity of modeling parameters affects the simulation's fidelity. These parameters should be calibrated and fine-tuned by reproducing the actual demolition data or video recordings. Some of these parameters are Blender's internal parameters that need to be found through some trial and error process because some of them are numerical artifacts. Table 1

lists Blender's internal parameters as well as the physical parameters. More details of Blender's internal parameters can be found in [22].

Table 1. List of simulation parameters

Parameter	**Category**
Wrecking Ball Mass	External
Bridge Total Mass	External
Bridge Model Geometry	External
Amounts of Explosives	External
Explosion Source Points	External
Source Limit (Number of fractures)	Cell Fracture
Noise	Cell Fracture
Neighbor Limit	Bullet Constraints Tool
Search Radius	Bullet Constraints Tool
Friction Coefficient	Bullet Constraints Tool
Bounciness	Bullet Constraints Tool
Collision Margin	Bullet Constraints Tool
Activation Linear Velocity	Bullet Constraints Tool
Activation Angular Velocity	Bullet Constraints Tool
Constraints Type	Bullet Constraints Tool
Constraint Breaking Threshold	Bullet Constraints Tool
Rigid Body Type	Rigid Body
Collision Shape	Rigid Body Collisions
Field_Hide	Force Field
Field_Type	Force Field
Field_Z_Direction	Force Field
Field_Shape	Force Field
Field_Strength	Force Field
Field_Flow	Force Field
Field_Seed	Force Field

Field_Apply_To_Location	Force Field
Field_Apply_To_Rotation	Force Field
Field_Falloff_Power	Force Field
Field_Falloff_Type	Force Field
Field_Use_Min_Distance	Force Field
Field_Use_Max_Distance	Force Field
Field_Distance_Max	Force Field
Field_Use_Radial_Min	Force Field
Field_Use_Radial_Max	Force Field
Field_Radial_Max	Force Field
Field_Radial_Falloff	Force Field
Rotation_90	Force Field

A set of simulations have been carried out to reproduce the I-235 Bridge demolition project. The bridge is modeled with the actual geometry (such as span length, deck width, pier height, etc.), then material properties are then entered, and discretized in Blender. The demolition of I-235 Bridge demolition was done by utilizing a wrecking ball, for which three set of simulations are carried out for one, two, and three span bridges, respectively. Figure 60 and 61 show the simulation of the one span bridge demolition after the other decks already destructed. Figure 62 and 63 show the visualized demolition simulation. There are a set of physics modeling parameters need to be calibrated for realistic simulations. Table 2 shows the key parameter values used for the reasonable simulation fidelity. Figure 64 and 65 show the I-235 Bridge with two spans remaining with the other decks destructed. Figure 66 and 67 show the corresponding visualization performed using the exactly same set of physics modeling parameters calibrated with the demolition simulation of single span I-235 Bridge (Figure 60 and 61). The impact of the location on the three-span bridge is then simulated where the influence of different impact points of the wrecking ball is also estimated. Four different simulations are performed for four different impact locations as shown in Figure 68 to 71.

Figure 60. I-235 Bridge with a span about to be demolished by dropping a wrecking ball

Figure 61. The one last remaining deck of I-235 Bridge demolished by the wrecking ball

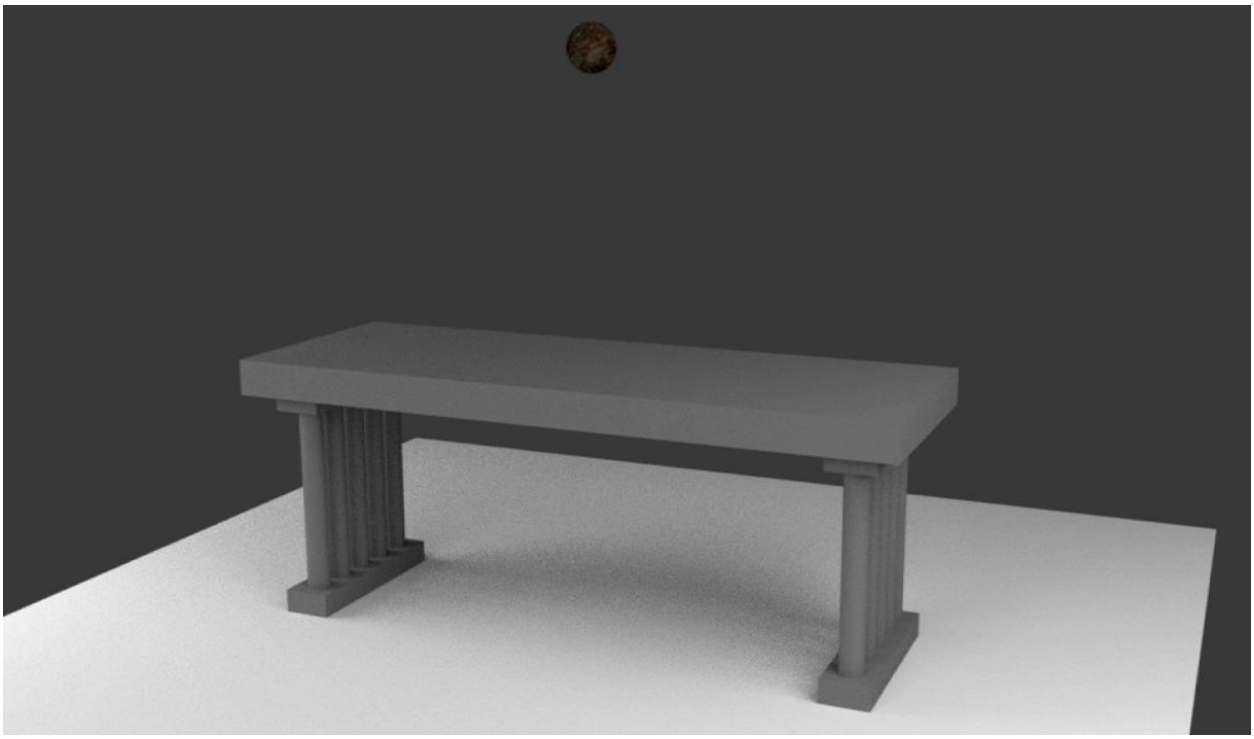

Figure 62. The one span I-235 Bridge modeled

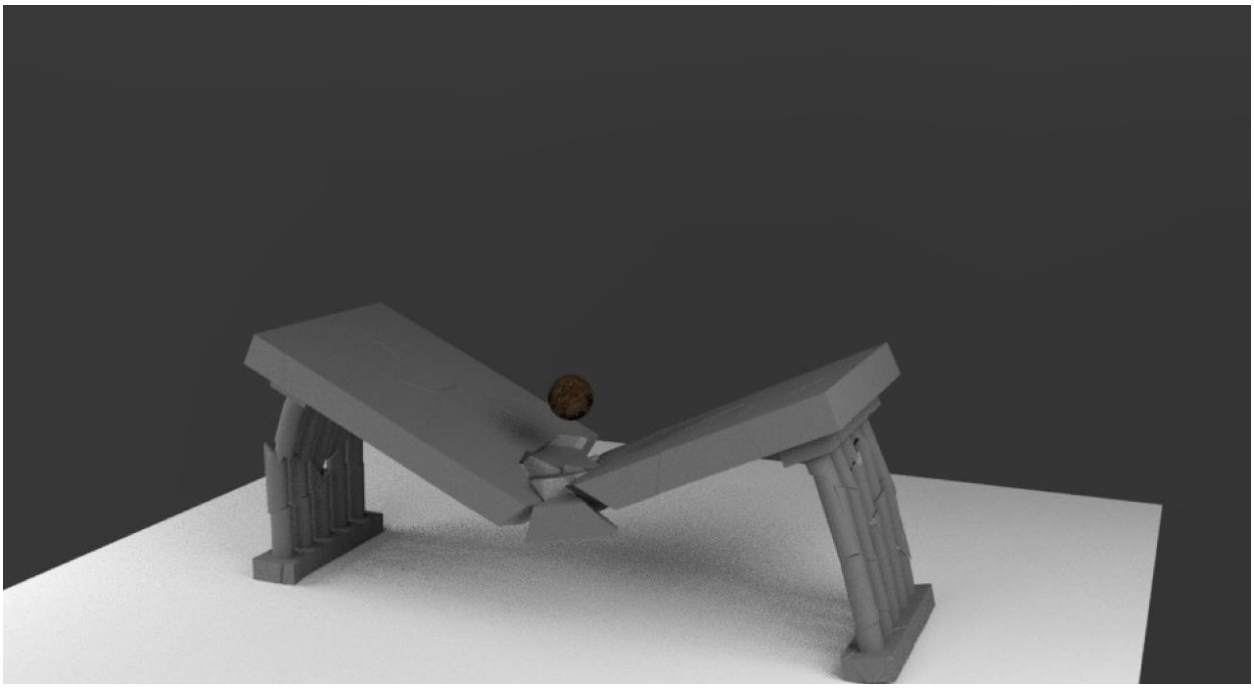

Figure 63. Simulated demolition of the one last remaining deck of I-235 Bridge

Table 2.List of key parameter values for the demolition simulation of I-235 Bridge

Parameter	Values
Bridge Mass (single span)	107,000 kg
Wrecking Ball Mass	10,000 kg
Fracturing Algorithm	Voronoi Boolean
Shard Count	200
Constraint Method	Vertex
Constraint Type	Hinge
Constraint Limit per mesh island	50
Search Radius	1.00 m
Constraint Breaking Threshold	10000 kgf

Figure 64. I-235 Bridge with two spans about to be demolished by dropping a wrecking ball

Figure 65. The one of the remaining decks of I-235 Bridge demolished by the wrecking ball

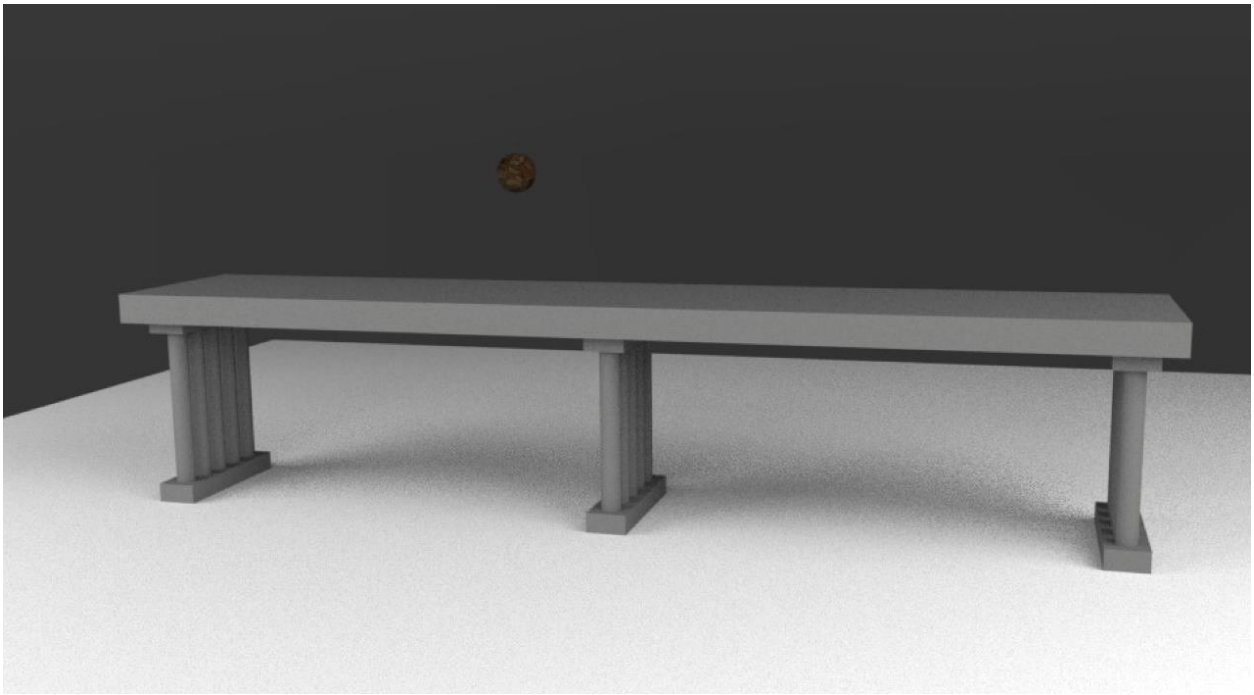

Figure 66. The two span I-235 Bridge modeled

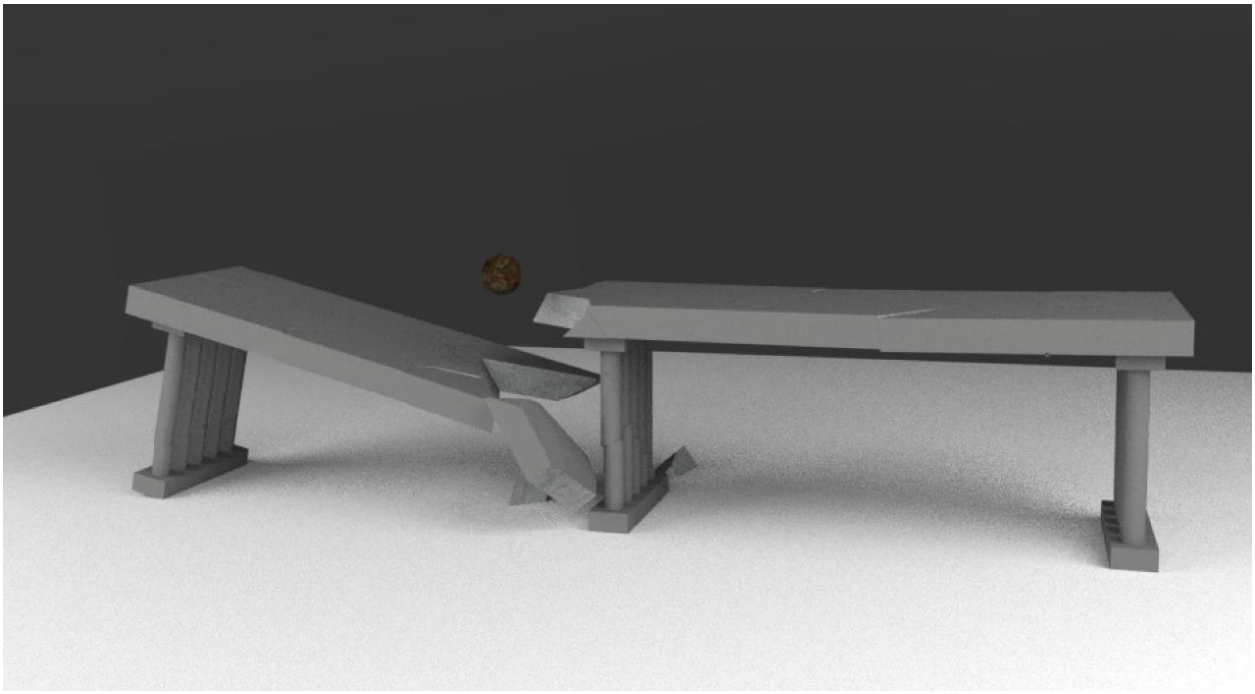

Figure 67. Simulated demolition of the one of remaining decks in I-235 Bridge

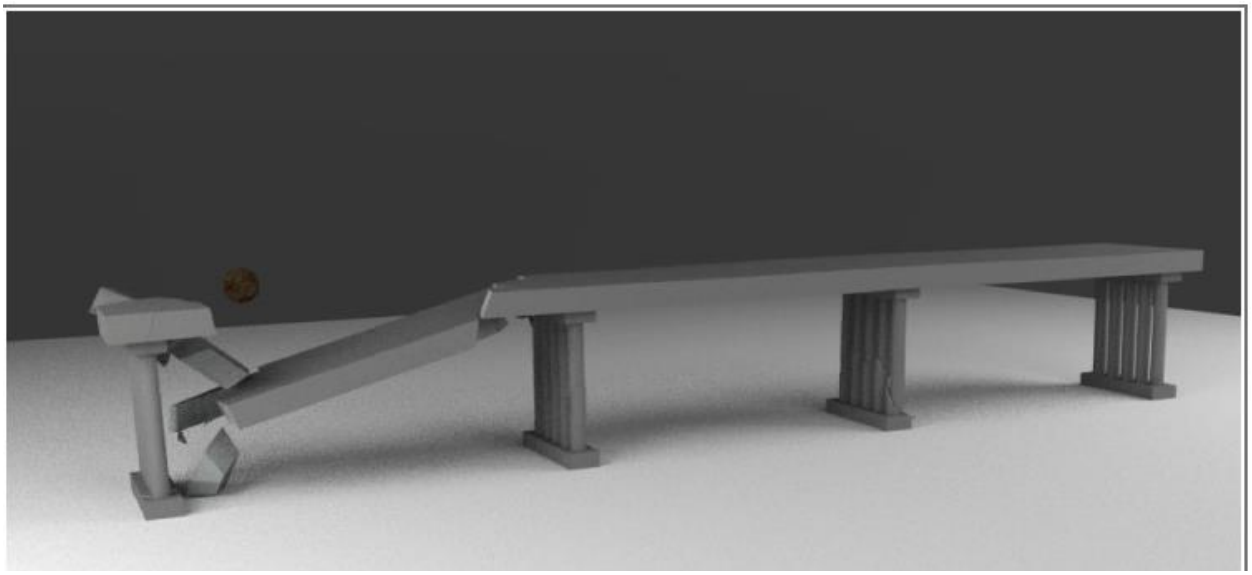

Figure 68. Simulated demolition of the three span I-235 Bridge, where the wrecking ball is dropped near the end of the bridge

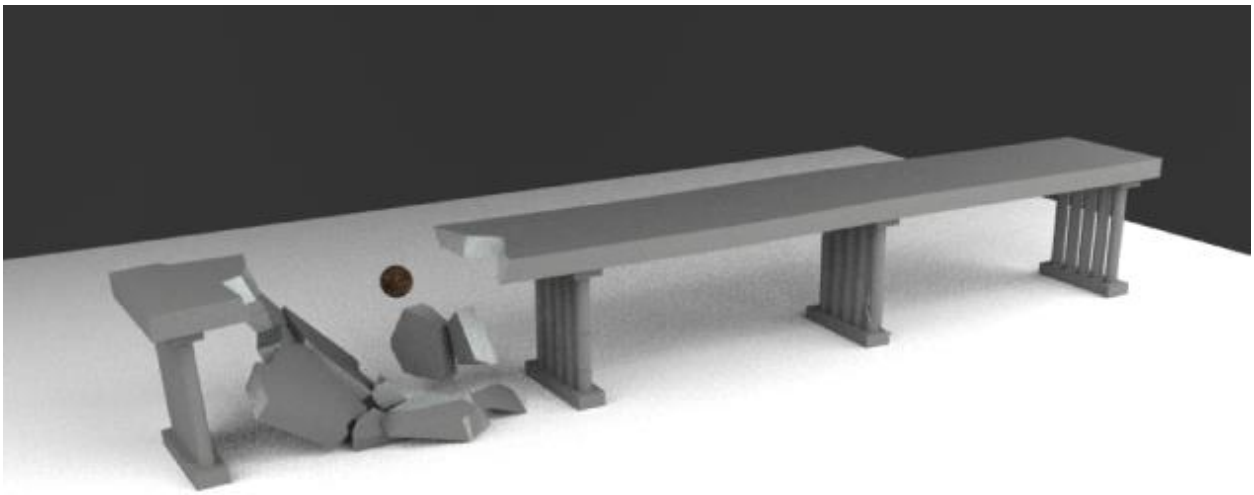

Figure 69. Simulated demolition of the three span I-235 Bridge, where the wrecking ball is dropped near the center of the neighboring bridge deck.

.

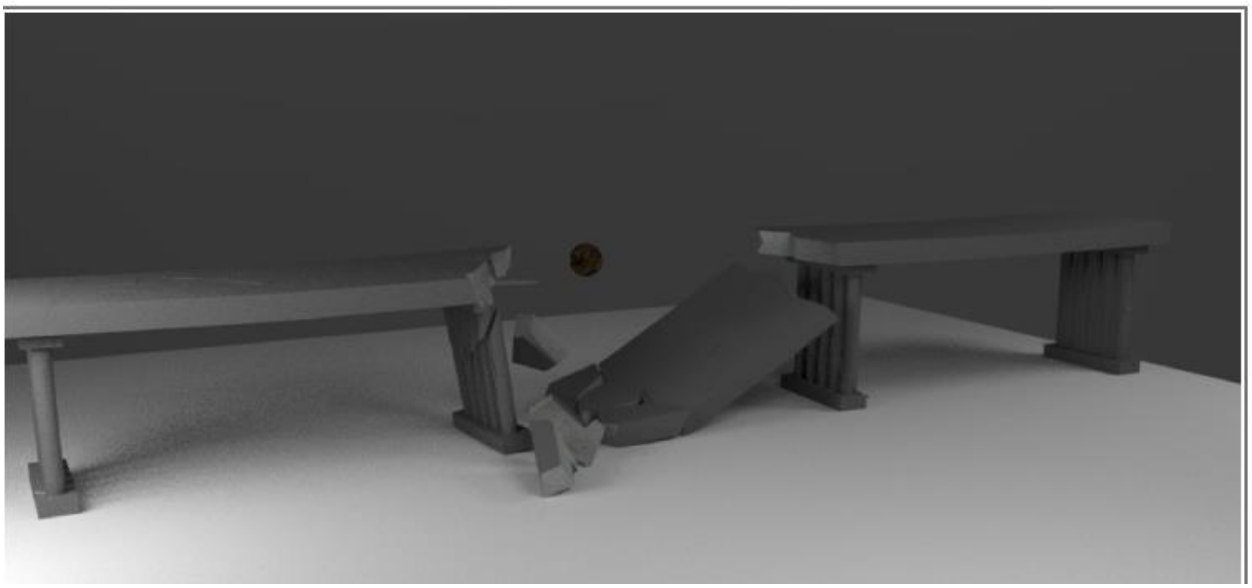

Figure 70. Simulated demolition of the three span I-235 Bridge, where the wrecking ball is dropped on the mid bridge deck; the hitting point is between the center of the mid bridge deck and the neighboring deck on the left in the figure.

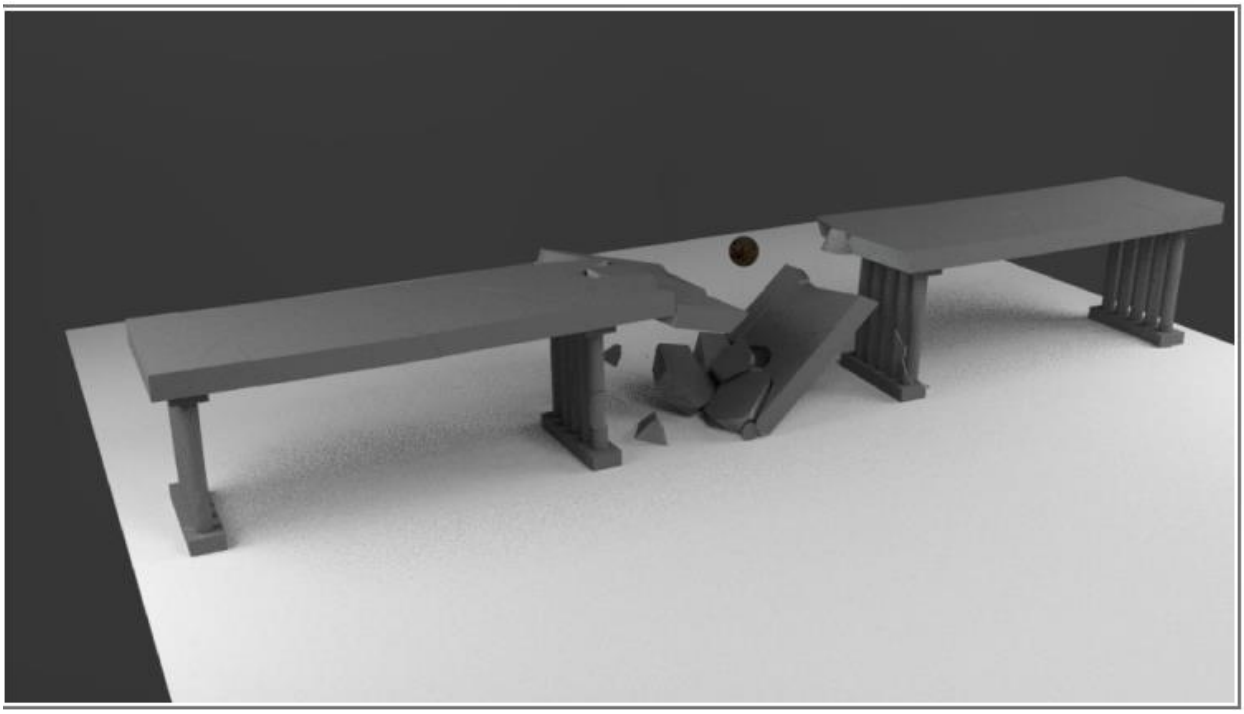

Figure 71. Simulated demolition of the three span I-235 Bridge, where the wrecking ball is dropped at the center of mid bridge.

Figure 72 shows the dismantling demolition using a jackhammer. The corresponding simulation is shown in Figure 73 that presents a good level of realism. Reproduction of this type of demolition requires relatively fine mesh compared to the wrecking ball demolition simulation above, which is therefore computationally more expensive in general.

Figure 72. Dismantling demolition using a jackhammer

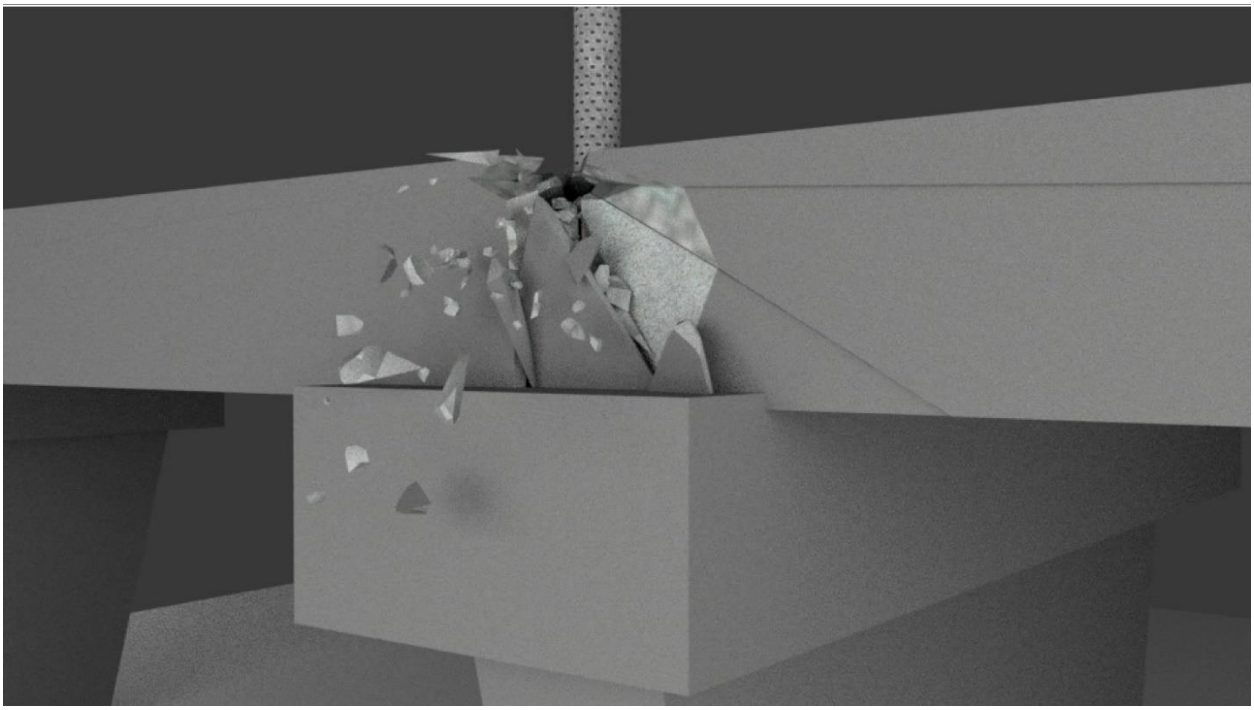

Figure 73. Simulated demolition by jackhammer

Chapter 3. CONCLUSION AND RECOMMENDATIONS

This report summarizes the research activities undertaken through an ABC-UTC project – Predictive Computer Program for Proactive Demolition Planning and introduces the developed framework to numerically predict various bridge demolition scenarios by two popular methods: explosion and using wrecking ball. The framework leverages Blender, a free and opensource software for visualizing demolition simulation and utilizes Bullet Physics, another free and open source physics engine library, to numerically model the simulation to implement demolition methods. Bridge Demolition Add-on (BDA) is developed as an add-on to these existing programs to add the new features that are not currently available such as contact forces retrieval. BDA also helps simulate the demolition by explosives and enhance the post-processing of the results through improved visualization.

This report provides an alternative approach to the existing empirical methods to plan the demolition such that engineers can leverage the software package to proactively plan the inherently risky and dangerous project. Therefore, the outcomes of this project will enable the decision-makers get better prepared for the worst-case scenarios and choose the best demolition planning ahead depending on the target structure's configuration and condition as well as the environmental considerations, time limits, budget, by proactively performing the visualized demolition scenarios. This document and software package will be of interest to highway officials, bridge (de)constructors, and structural engineers, as well as other decision-makers concerned with the bridge demolition. This project outcome will lead to minimizing demolition project costs and minimizing adverse risks inherent in demolition projects and contribute to maintain safety of workers and surrounding environment and infrastructures.

While the software package developed in this document provides a promising alternative for proactive demolition planning to optimize the cost and enhance the safety, the programs in the package was not calibrated to the existing demolition data due to the limitation of project duration and budget. Therefore, the software needs to be further enhanced with calibration such that it can be used as a quantitative tool to improve the predictive capability to estimate the actual field demolitions.

REFERENCES

[1] D. Garber, “Compilation of Results from Bridge Demolition DOT Survey,” Miami, FL, 2016.

[2] D. Garber, “Compilation of Results from Bridge Demolition DOT Survey,” Miami, FL, 2016.

[3] T. D. of Transportation, “Guidelines for Preparation of Demolition Plans for Structures Over Railroads, Burlington Northern Santa Fe Railroad.”

[4] J. Ghaboussi, “Fully deformable discrete element analysis using a finite element approach,” *Comput. Geotech.*, 1988.

[5] F. A. Tavarez and M. E. Plesha, “Discrete element method for modelling solid and particulate materials,” *Int. J. Numer. Methods Eng.*, vol. 70, no. 4, pp. 379–404, Apr. 2007.

[6] G. A. D’Addetta, *Discrete Models for Cohesive Frictional Materials*. University of Stuttgart, 2004.

[7] E. Masoero, F. K. Wittel, H. J. Herrmann, and B. M. Chiaia, “Progressive Collapse Mechanisms of Brittle and Ductile Framed Structures,” *J. Eng. Mech.*, vol. 136, no. 8, pp. 987–995, Aug. 2010.

[8] S. J. Lee and Y. M. A. Hashash, “iDEM: An impulse-based discrete element method for fast granular dynamics,” *Int. J. Numer. Methods Eng.*, 2015.

[9] E. Coumans, “Bullet Physics.” 2017.

[10] F. Aurenhammer, “Voronoi diagrams---a survey of a fundamental geometric data structure,” *ACM Comput. Surv.*, 2002.

[11] D. Zhao, E. G. Nezami, Y. Hashash, and J. Ghaboussi.J., “Three-dimensional discrete element simulation for granular materials,” *Comput. Eng. Comput. Int. J. Eng. Softw.*, vol. 23, no. 7, pp. 749–770, 2006.

[12] The Blender Foundation, “Blender 2.77.0 Python API Documentation.”

[13] The Blender Foundation, “Blender 2.78.0 Python API Documentation.”

[14] The Blender Foundation, “Blender 2.79.0 Python API Documentation.”

[15] The Blender Foundation, “Blender.” 2015.

[16] K. Kostack, “Deformation Visualizer v1.02 by Kai Kostack.” .

[17] G. F. Kinney and K. J. Graham, *Explosive Shocks in Air*. 2013.

[18] E. Coumans, *Bullet 2.80 Physics SDK Manual*. Bullet Physics Library, 2012.

[19] N. S. (TOPTAL), “Video Game Physics Tutorial Part II: Collision Detection.”

[20] C. Ericson, *Real-Time Collision Detection*, no. 1. 2005.

[21] E. Coumans, “Bullet Physics.” 2017.

[22] C. of Blender, “Blender User Manual,” 2015.

[23] The Blender Foundation, “Blender Fracture Modifier.” .

[24] Bashi, “Bullet Viewport Constraints Tool.” .

APPENDIX A: IMPLMENTED CODES

Developed codes are provided as appendix as below. Codes are implemented in Python for developing Bridge Demolition Add-on (v 1.0.0). Codes developed for contour visualization and renderer are based on [8].

```
bl_info = {
    'name': 'Bridge Demolition',
    "author": "Ali Bakhtiari",
    "version": (1, 0, 0),
    "blender": (2, 79, 0),
    "category": "All",
    "description": "Bridge Demolition Add On"
}



#############################################################################

import bpy, bmesh, mathutils, sys, math, struct, time
from bpy import context
from mathutils import Vector
from mathutils import Vector
from mathutils import Color

#############################################################################
# Global Variables
USING_CONTOUR = True

### Vars
qRenderAnimation = 1
displaySteps = 200
fadeDecrement = 2

materialName = "ContourGradient_"

# source point of explosion
SOURCE_POINT = Vector((0,-20,7))

```

```
# Field Force Properties
FIELD_HIDE = True
FIELD_TYPE = 'FORCE'
FIELD_Z_DIRECTION = 'BOTH'
FIELD_SHAPE = 'POINT' #'PLANE'
FIELD_STRENGTH = 0.0
FIELD_FLOW = 0.0
FIELD_SEED = 40
FIELD_APPLY_TO_LOCATION = True
FIELD_APPLY_TO_ROTATION = True
FIELD_FALLOFF_TYPE = 'TUBE'
FIELD_FALLOFF_POWER = 10.0
FIELD_USE_MIN_DISTANCE = True
FIELD_USE_MAX_DISTANCE = False
FIELD_DISTANCE_MAX = 5 # 0.2
FIELD_USE_RADIAL_MIN = False
FIELD_USE_RADIAL_MAX = True
FIELD_RADIAL_MAX = 2 # 0.2
FIELD_RADIAL_FALLOFF = 0.0 # real graviational falloff = 2!!

# Rotation
ROTATION_90 = 1.5707961320877075


################################################################################
# Panel UI
################################################################################

class CustomDrawOperator(bpy.types.Operator):
    bl_idname = "object.custom_draw"
    bl_label = "Import"

    filepath = bpy.props.StringProperty(subtype="FILE_PATH")

    my_float = bpy.props.FloatProperty(name="Float")
    my_bool = bpy.props.BoolProperty(name="Toggle Option")
    my_string = bpy.props.StringProperty(name="String Value")

    def execute(self, context):
        print("filepath:", self.filepath)
        bpy.context.scene.explosion_tool.debrisfilepath = self.filepath
        return {'FINISHED'}

```

```
    def invoke(self, context, event):
        context.window_manager.fileselect_add(self)
        return {'RUNNING_MODAL'}

    def draw(self, context):
        layout = self.layout
        col = layout.column()
        col.label(text="Custom Interface!")

        row = col.row()
        row.prop(self, "my_float")
        row.prop(self, "my_bool")

        col.prop(self, "my_string")


class CustomDrawOperator2(bpy.types.Operator):
    bl_idname = "object.custom_draw2"
    bl_label = "Import"

            filepath = bpy.props.StringProperty(subtype="FILE_PATH")

            my_float = bpy.props.FloatProperty(name="Float")
            my_bool = bpy.props.BoolProperty(name="Toggle Option")
            my_string = bpy.props.StringProperty(name="String Value")

            def execute(self, context):
                print("filepath:", self.filepath)
                bpy.context.scene.explosion_tool.logfilepath = self.filepath
                return {'FINISHED'}

            def invoke(self, context, event):
                context.window_manager.fileselect_add(self)
                return {'RUNNING_MODAL'}

            def draw(self, context):
                layout = self.layout
                col = layout.column()
                col.label(text="Custom Interface!")

                row = col.row()
                row.prop(self, "my_float")
                row.prop(self, "my_bool")

```

```
            col.prop(self, "my_string")


    class MessageBox(bpy.types.Operator):
        bl_idname = "message.messagebox"
        bl_label = ""

        message = bpy.props.StringProperty(
            name = "message",
            description = "message",
            default = ''
        )

        def execute(self, context):
            self.report({'INFO'}, self.message)
            print(self.message)
            return {'FINISHED'}

        def invoke(self, context, event):
            return context.window_manager.invoke_props_dialog(self, width = 400)

        def draw(self, context):
            self.layout.label(self.message)
            self.layout.label("")



    class SceneItems(bpy.types.PropertyGroup):
        source = bpy.props.FloatVectorProperty()

    class AmountItems(bpy.types.PropertyGroup):
        amount = bpy.props.FloatProperty()

    class DebrisLogFileOperator(bpy.types.Operator):
        bl_idname = "scene.debris_log_file_operator"
        bl_label = "Set Debris Log Filepath"

        def execute(self, context):
            bpy.ops.object.custom_draw('INVOKE_DEFAULT')
            return {'FINISHED'}

    class ForceLogFileOperator(bpy.types.Operator):
        bl_idname = "scene.logfilepath_operator"
```

```
169.            bl_label = "Set Force Log Filepath"
170.
171.            def execute(self, context):
172.                bpy.ops.object.custom_draw2('INVOKE_DEFAULT')
173.                return {'FINISHED'}
174.
175.
176.        class MassButtonOperator(bpy.types.Operator):
177.            bl_idname = "scene.mass_button_operator"
178.            bl_label = "Show Total Mass of Particles"
179.
180.            def execute(self, context):
181.                msg = 'Total Mass: ' + str(total_mass())
182.                bpy.ops.message.messagebox('INVOKE_DEFAULT', message = msg)
183.                return {'FINISHED'}
184.
185.
186.        class AddButtonOperator(bpy.types.Operator):
187.            bl_idname = "scene.add_button_operator"
188.            bl_label = "Add Source Point"
189.
190.            def execute(self, context):
191.                id = len(context.scene.collection)
192.                new = context.scene.collection.add()
193.                new.name = str(id)
194.                new.source = (0,0,0)
195.                newa = context.scene.amountcollection.add()
196.                newa.amount = 0
197.                return {'FINISHED'}
198.
199.        class BuildButtonOperator(bpy.types.Operator):
200.            bl_idname = "scene.build_button_operator"
201.            bl_label = "Build Force Objects"
202.
203.            def execute(self, context):
204.
205.                build_all_force_objects()
206.                return {'FINISHED'}
207.
208.
209.
210.        class ButtonOperator(bpy.types.Operator):
211.            bl_idname = "scene.button_operator"
212.            bl_label = "Button"
```

```
213.
214.           id = bpy.props.StringProperty()
215.
216.           def execute(self, context):
217.               print("Pressed button ", self.id)
218.               for item in context.scene.collection:
219.                   print(item.value[0])
220.               return {'FINISHED'}
221.
222.       class DeleteOperator(bpy.types.Operator):
223.           bl_idname = "scene.delete_operator"
224.           bl_label = "Delete"
225.
226.           ids = bpy.props.IntProperty()
227.
228.           def execute(self, context):
229.               context.scene.collection.remove(self.ids)
230.               context.scene.amountcollection.remove(self.ids)
231.               return {'FINISHED'}
232.
233.       class FancyPanel(bpy.types.Panel):
234.           bl_idname = "panel.panel_fancy_panel"
235.           bl_label = "Explosion"
236.           bl_space_type = "VIEW_3D"
237.           bl_region_type = "TOOLS"
238.           bl_category = "BDA"
239.
240.
241.           def draw(self, context):
242.               layout = self.layout
243.               row = layout.row()
244.               row.prop(bpy.context.scene.explosion_tool, "explosion_flag")
245.               self.layout.operator("scene.add_button_operator")
246.               row = layout.row()
247.               i = 0
248.               for item in context.scene.collection:
249.                  row = self.layout.row(align=True)
250.                  row.prop(item, "source", text="source " + str(i))
251.                  row.prop(context.scene.amountcollection[i], "amount")
252.                  row.operator("scene.delete_operator", text="Delete").ids = i
253.                  i = i + 1
254.
255.               row = layout.row(align=True)
256.               row.operator("scene.build_button_operator")
```

```
            row = layout.row()
            self.layout.operator("scene.mass_button_operator")


    # ------------------------------------------------------------------------
    #    Scene Properties
    # ------------------------------------------------------------------------

    class ExplosionVariables(bpy.types.PropertyGroup):
        logfilepath = bpy.props.StringProperty(
            name = "force filepath",
            description = "write to file total contact force values"
        )
        debrisfilepath = bpy.props.StringProperty(
            name = "debris filepath",
            description = "write to file debris propagation data"
        )
        contour_flag = bpy.props.BoolProperty(
            name="Contours",
            description="Show total contact force contours",
            default=True
        )
        explosion_flag = bpy.props.BoolProperty(
            name="Explosion",
            description="Activate Explosion",
            default=False
        )

    class ExplosionPanel(bpy.types.Panel):
        bl_idname = "panel.explosion_panel"
        bl_label = "Post Processing"
        bl_space_type = "VIEW_3D"
        bl_region_type = "TOOLS"
        bl_category = "BDA"


        def draw(self, context):
            layout = self.layout
            scene = context.scene
            explosion_tool = scene.explosion_tool
```

```
301.                # Create an row where the buttons are aligned to each other.
302.                layout.label(text=" Contact Force")
303.                row = layout.row()
304.                row.prop(bpy.context.scene.explosion_tool, "contour_flag")
305.                row = layout.row()
306.                self.layout.operator("scene.debris_log_file_operator")
307.                self.layout.prop(bpy.context.scene.explosion_tool, 'debrisfilepath')
308.
309.                row = layout.row()
310.
311.                self.layout.operator("scene.logfilepath_operator")
312.                self.layout.prop(bpy.context.scene.explosion_tool, 'logfilepath')
313.
314.        class RegisterHandlerOperator(bpy.types.Operator):
315.            bl_idname = "scene.register_handler_operator"
316.            bl_label = "Register Handler"
317.
318.            def execute(self, context):
319.                print('Initializing event handler.')
320.                bpy.app.handlers.frame_change_pre.append(main)
321.                return {'FINISHED'}
322.
323.        class UnregisterHandlerOperator(bpy.types.Operator):
324.            bl_idname = "scene.unregister_handler_operator"
325.            bl_label = "Unregister Handler"
326.
327.            def execute(self, context):
328.                print('Removing event handler.')
329.                for i in range( len( bpy.app.handlers.frame_change_pre ) ):
330.                    bpy.app.handlers.frame_change_pre.pop()
331.                return {'FINISHED'}
332.
333.        class HandlerPanel(bpy.types.Panel):
334.            bl_idname = "panel.handler_panel"
335.            bl_label = "Frame Change Event Handler"
336.            bl_space_type = "VIEW_3D"
337.            bl_region_type = "TOOLS"
338.            bl_category = "BDA"
339.
340.            def draw(self, context):
341.                layout = self.layout
342.                scene = context.scene
343.
344.                self.layout.operator("scene.register_handler_operator")
```

```
345.                self.layout.operator("scene.unregister_handler_operator")
346.
347.
348.
349.        class ExplosionOperator (bpy.types.Operator):
350.            bl_idname = "wm.explosion_operator"
351.            bl_label = "Add Source Point"
352.
353.            def execute(self, context):
354.                scene = context.scene
355.                explosion_tool = scene.explosion_tool
356.
357.                i = 0
358.                for item in context.scene.collection:
359.                   print(item.source[0], item.source[1], item.source[2], context.scene.amountcollection[i].amount)
360.                   i = i + 1
361.                return {'FINISHED'}
362.
363.
364.        ################################################################################
365.        # Contour & Explosion
366.        ################################################################################
367.
368.
369.
370.        def total_mass():
371.                total_mass = 0.0
372.                for obj in bpy.data.objects:
373.                    if obj.type != 'MESH':
374.                        continue
375.
376.                    if not hasattr(obj.rigid_body, 'type'):
377.                        continue
378.
379.                    if obj.rigid_body.type != 'ACTIVE':
380.                        continue
381.
382.                    # create_empty_obj
383.                    total_mass = total_mass + obj.rigid_body.mass
384.
385.                return total_mass
386.
387.        ################################################################################
388.
```

```
389.        def log_debris_locations():
390.            if (not bpy.context.scene.explosion_tool.debrisfilepath):
391.                bpy.context.scene.explosion_tool.debrisfilepath = "debrisfilepath.log"
392.            f = open(bpy.context.scene.explosion_tool.debrisfilepath, "a+")
393.            for obj in bpy.data.objects:
394.                if obj.type != 'MESH':
395.                        continue
396.                if not hasattr(obj.rigid_body, 'type'):
397.                    continue
398.
399.                if obj.rigid_body.type != 'ACTIVE':
400.                    continue
401.                f.write(obj.name + ":" + str(obj.rigid_body.location[0]) + "," + str(obj.rigid_body.loca-
    tion[1]) + "," + str(obj.rigid_body.location[2]) + "\n")
402.
403.            f.close()
404.
405.
406.        ############################################################################
407.
408.        def build_all_force_objects():
409.            for obj in bpy.data.objects:
410.                if obj.type != 'MESH':
411.                    continue
412.
413.                if not hasattr(obj.rigid_body, 'type'):
414.                    continue
415.
416.                if obj.rigid_body.type != 'ACTIVE':
417.                    continue
418.
419.                # create_empty_obj
420.                create_empty_obj(obj, 0)
421.                create_empty_obj(obj, 1)
422.                create_empty_obj(obj, 2)
423.
424.        ############################################################################
425.        # create empty object
426.
427.        def create_empty_obj(parent_obj, axes_index):
428.            emptyname = 'Empty_' + str(axes_index) + '_' + parent_obj.name
429.
430.            if bpy.data.objects.get(emptyname) is None:
431.                bpy.ops.object.empty_add(type='PLAIN_AXES')
```

```
                emptyobj = bpy.data.objects[bpy.context.active_object.name]
                emptyobj.name = emptyname
                emptyobj.select = False
                emptyobj.hide = FIELD_HIDE
                #emptyobj.parent = parent_obj
                emptyobj.location = parent_obj.location
                emptyobj.rotation_euler[0] = 0
                emptyobj.rotation_euler[1] = 0
                emptyobj.rotation_euler[2] = 0
                if (axes_index < 2):
                    emptyobj.rotation_euler[axes_index] = ROTATION_90

                emptyobj.field.type = FIELD_TYPE
                emptyobj.field.z_direction = FIELD_Z_DIRECTION
                emptyobj.field.shape = FIELD_SHAPE
                emptyobj.field.strength = FIELD_STRENGTH
                emptyobj.field.flow = FIELD_FLOW
                emptyobj.field.seed = FIELD_SEED
                emptyobj.field.apply_to_location = FIELD_APPLY_TO_LOCATION
                emptyobj.field.apply_to_rotation = FIELD_APPLY_TO_ROTATION
                emptyobj.field.falloff_type = FIELD_FALLOFF_TYPE
                emptyobj.field.falloff_power = FIELD_FALLOFF_POWER
                emptyobj.field.use_min_distance = FIELD_USE_MIN_DISTANCE
                emptyobj.field.use_max_distance = FIELD_USE_MAX_DISTANCE
                emptyobj.field.distance_max = FIELD_DISTANCE_MAX
                emptyobj.field.use_radial_min = FIELD_USE_RADIAL_MIN
                emptyobj.field.use_radial_max = FIELD_USE_RADIAL_MAX
                emptyobj.field.radial_max = FIELD_RADIAL_MAX
                emptyobj.field.radial_falloff = FIELD_RADIAL_FALLOFF


        ################################################################################
        def update_empty_objects(parent_obj, explosion_force):
            for i in range(3):
                emptyobj = bpy.data.objects["Empty_" + str(i) + "_" + parent_obj.name]
                emptyobj.location = parent_obj.location
                #emptyobj.field.strength = explosion_force[i]
                if i < 2:
                    emptyobj.field.strength = explosion_force[abs(i-1)]
                else :
                    emptyobj.field.strength = explosion_force[i]


        ################################################################################
```

```
476.
477.       # explosion
478.
479.       def explosion(m_origin, m_source, time_elapsed, projected_area, amount):
480.           Distance = [0,0,0] #btVector3
481.
482.           w = amount;
483.
484.           Distance[0] = m_origin[0] - m_source[0]
485.           Distance[1] = m_origin[1] - m_source[1]
486.           Distance[2] = m_origin[2] - m_source[2]
487.
488.           distance = pow((Distance[0] * Distance[0] + Distance[1] * Distance[1] + Distance[2] * Distance[2]), 0.5);
489.
490.           distance /= pow(w,1/3);
491.
492.
493.           area = projected_area;
494.
495.           ambientPressure = 101325.0;
496.
497.           peakOverpressure = ambientPressure * 808 * ((pow(distance / 4.5, 2) + 1) / (pow((1 + (pow(dis-
    tance / 0.048, 2))), 0.5) * pow((1 + (pow(distance / 0.32, 2))), 0.5) * pow((1 + (pow(distance / 1.35, 2))), 0.5)));
498.
499.           millisecond_to_second = 0.001
500.
501.           td = millisecond_to_second * 980 * (1 + pow(distance / 0.54, 10) / ((1 + pow(dis-
    tance / 0.02, 3.0)) * (1 + pow(distance / 0.74, 6)) * (pow(1 + pow(distance / 6.9, 2), 0.5))) * pow(w, 1/3));
502.
503.           beta = -1.0;
504.
505.           #Friedlander Equation
506.           pressure = ambientPressure + peakOverpressure * (1 - time_elapsed / td) * math.exp(beta * time_elapsed / td);
507.
508.           explosionforcex = pressure * area * Distance[0] / pow((Distance[0] * Distance[0] + Distance[1] * Dis-
    tance[1] + Distance[2] * Distance[2]), 0.5);
509.           explosionforcey = pressure * area * Distance[1] / pow((Distance[0] * Distance[0] + Distance[1] * Dis-
    tance[1] + Distance[2] * Distance[2]), 0.5);
510.           explosionforcez = pressure * area * Distance[2] / pow((Distance[0] * Distance[0] + Distance[1] * Dis-
    tance[1] + Distance[2] * Distance[2]), 0.5);
511.
512.           m_explosionForce = [explosionforcex, explosionforcey, explosionforcez];
513.           return m_explosionForce;
514.
```

```
515.
516.
517.        ###########################################################################
518.        # bmesh from object
519.
520.        def bmesh_from_object(ob):
521.            matrix = ob.matrix_world
522.            me = ob.to_mesh(bpy.data.scenes['Scene'], apply_modifiers=True, settings='PREVIEW')
523.            me.transform(matrix)
524.            bm = bmesh.new()
525.            bm.from_mesh(me)
526.            bpy.data.meshes.remove(me)
527.            return bm
528.
529.        ###########################################################################
530.        # dot product
531.
532.        def dot_product(vec1, vec2):
533.            sum = vec1[0]*vec2[0]+vec1[1]*vec2[1]+vec1[2]*vec2[2]
534.            return sum
535.
536.        ###########################################################################
537.        # projected area
538.
539.        def projected_area(obj, projection_normal_vector):
540.            bm = bmesh_from_object(obj)
541.            bm.normal_update()
542.            print('bmesh:', bm)
543.
544.            pa = 0
545.            for f in bm.faces:
546.                face_area = f.calc_area()
547.                face_normal = f.normal
548.
549.                # compute projected area
550.                face_pa = abs(dot_product(projection_normal_vector, face_normal)) * face_area
551.                pa += face_pa
552.                print('face index:', f.index, ' and face area:', face_area, 'face normal:', face_nor-
    mal, 'face pa:', face_pa)
553.                print
554.
555.            bm.free()
556.            pa = pa/2
557.            return pa
```

```
558.
559.
560.
561.
562.        ###############################################################################
563.
564.        def do_explosion(current_frame):
565.            print("do:", current_frame)
566.            # deselect all of the objects
567.            bpy.ops.object.select_all(action='DESELECT')
568.
569.            time_scale = bpy.context.scene.rigidbody_world.time_scale
570.            steps_per_second = bpy.context.scene.rigidbody_world.steps_per_second
571.
572.            time_elapsed = current_frame * time_scale / steps_per_second
573.
574.            if (not bpy.context.scene.explosion_tool.logfilepath):
575.                bpy.context.scene.explosion_tool.logfilepath = "forcelogfile_explosion.log"
576.            f = open(bpy.context.scene.explosion_tool.logfilepath+"_explosion.log", "a+")
577.            f.write('name, projected area, explosion force, projection_normal_vector\n')
578.            f.write('current_frame: ' + str(current_frame) + 'elapsed time: ' + str(time_elapsed) + '\n')
579.            for obj in bpy.data.objects:
580.                if obj.type != 'MESH':
581.                    continue
582.
583.                if not hasattr(obj.rigid_body, 'type'):
584.                    continue
585.
586.                if obj.rigid_body.type != 'ACTIVE':
587.                    continue
588.
589.                projection_normal_vector = obj.rigid_body.location - SOURCE_POINT
590.
591.                pa = projected_area(obj, projection_normal_vector)
592.
593.                index = 0
594.                total_explosion_force = [0, 0, 0]
595.                for item in bpy.context.scene.collection:
596.                    amount = bpy.context.scene.amountcollection[index].amount
597.                    print("amount", amount)
598.                    explosion_force = explosion(obj.rigid_body.location, item.source, time_elapsed, pa, amount)
599.                    print("explosion_force: ", explosion_force)
600.                    for i in range(3):
601.                        total_explosion_force[i] = total_explosion_force[i] + explosion_force[i]
```

```
602.                     print("total_explosion_force: ", total_explosion_force)
603.                 index = index +1
604.
605.             print('name:', obj.name, 'projected area:', pa, 'explosion force:', total_explosion_force, 'projection_nor-
    mal_vector:', projection_normal_vector)
606.             f.write(obj.name + ',' + str(pa) + ',' + str(explosion_force) + ',' + str(projection_normal_vec-
    tor) + '\n')
607.
608.             # apply force
609.             update_empty_objects(obj, total_explosion_force)
610.
611.         f.close()
612.
613.     ################################################################################
614.
615.     def main(scene):
616.
617.
618.
619.         if scene.frame_current == scene.frame_end:
620.             log_debris_locations()
621.             if bpy.context.screen.is_animation_playing:
622.                 bpy.ops.screen.animation_play()
623.
624.
625.         if not "start-frame" in bpy.app.driver_namespace:
626.             em = bpy.app.driver_namespace["start-frame"] = []
627.             log_debris_locations()
628.
629.
630.         ############################
631.         ### What to do on start frame
632.
633.         if bpy.context.scene.explosion_tool.contour_flag:
634.
635.                 ### Render part
636.                 if qRenderAnimation:
637.
638.                     for i in range( len( bpy.app.handlers.frame_change_pre ) ):
639.                          bpy.app.handlers.frame_change_pre.pop()
640.
641.                     filepathOld = bpy.context.scene.render.filepath
642.                     bpy.context.scene.render.filepath += "%04d" %(scene.frame_current -1)
643.                     bpy.context.scene.render.image_settings.file_format = 'JPEG'
```

```
644.                         bpy.context.scene.render.image_settings.quality = 75
645.
646.
647.                         if qRenderAnimation == 1: bpy.ops.render.render(write_still=True)
648.
649.                         elif qRenderAnimation == 2: bpy.ops.render.opengl(write_still=True)
650.
651.                         bpy.context.scene.render.filepath = filepathOld
652.
653.
654.                         bpy.app.handlers.frame_change_pre.append(main)
655.
656.
657.                         ############################
658.                         ### What to do on start frame
659.                         if not "force-contour" in bpy.app.driver_namespace:
660.                             print("Initializing buffers...")
661.
662.
663.                             ###### Function
664.                             initBuffers(scene)
665.                             ###### Function
666.                             initMaterials()
667.                         else:
668.                             ###### Function
669.                             ################################
670.                             ### What to do AFTER start frame
671.                             print("Frame:", scene.frame_current)
672.
673.                             defObjs = bpy.app.driver_namespace["force-contour"]
674.
675.                             ###### Function
676.                             changeObjMaterials()
677.
678.
679.
680.             if bpy.context.scene.explosion_tool.explosion_flag:
681.                 if not "explosion" in bpy.app.driver_namespace:
682.                     print("Initializing buffers...")
683.                     ###### Function
684.                     initExplosion(scene)
685.                 else:
686.
687.                     defObjsExplosion = bpy.app.driver_namespace["explosion"]
```

```
                    do_explosion(scene.frame_current)


            ################################
            ### What to do AFTER start frame
            print("Frame:", scene.frame_current)




        ###############################################################################

        def initExplosion(scene):
            # Create property for frame change storage of data
            defObjsExplosion = bpy.app.driver_namespace["explosion"] = []


        ###############################################################################

        def initBuffers(scene):


            ### Make object list
            objs = []
            for obj in bpy.data.objects:
                # exclude constraint objects
                if obj.select and obj.type == 'MESH' and not obj.hide and obj.is_visible(scene) and not ha-
    sattr(obj.rigid_body_constraint, 'type'):
                    objs.append(obj)

            # Create property for frame change storage of data
            defObjs = bpy.app.driver_namespace["force-contour"] = []
            defObjsDict = bpy.app.driver_namespace["force-contour-dict"] = {}

            # frame force dictionary
            forceDict = {}

            # for all constraints calculate object forces
            objects = bpy.data.objects
            for obj in objects:
                if(hasattr(obj.rigid_body_constraint, 'type')):
```

```
731.
732.                     if obj.rigid_body_constraint.object1.name not in forceDict:
733.                         forceDict[obj.rigid_body_constraint.object1.name] = 0
734.
735.                     if obj.rigid_body_constraint.object2.name not in forceDict:
736.                         forceDict[obj.rigid_body_constraint.object2.name] = 0
737.
738.
739.             ### Create original transform data array
740.             d = -1
741.             for objname in forceDict:
742.                 d += 1
743.                 if d %100 == 0: sys.stdout.write('\r' +"%d" %d)
744.                 defObjForceDif = 0
745.                 defObjs.append([obj, forceDict[objname], defObjForceDif])
746.                 defObjsDict[objname] = d
747.                 ### Remove all materials but one from object
748.                 objectd = bpy.data.objects[objname]
749.                 bpy.context.scene.objects.active = objectd
750.
751.                 while len(objectd.material_slots) > 1:
752.                     objectd.active_material_index = 0
753.                     bpy.ops.object.material_slot_remove()
754.
755.
756.                 objectd["defLastStep"] = 0
757.
758.             print("defObjsDict", defObjsDict)
759.
760.         ################################################################################
761.
762.         def initMaterials():
763.
764.
765.
766.             ### Create gradient materials for later use and reuse
767.             for step in range(displaySteps):
768.                 dif = step *(1 /(displaySteps -1))
769.
770.                 col = Color((0, 0, 0))
771.                 #col.h = 0; col.s = 1; col.v = 1
772.                 difNormal = dif
773.                 col.r = difNormal   # correct math: = (difNormal -0.5) *2
774.                 col.g = 1 -(abs(0.5 -difNormal) *2)
```

```
775.                col.b = (0.5 -difNormal) *2
776.
777.                mat = bpy.data.materials.new(materialName +"%03d" %step)
778.                mat.diffuse_color = col
779.                col.s *= 0.5
780.                mat.specular_color = col
781.                mat.emit = 0.33
782.
783.
784.        ###############################################################################
785.
786.
787.        def changeObjMaterials():
788.
789.
790.
791.            time_scale = bpy.context.scene.rigidbody_world.time_scale
792.            steps_per_second = bpy.context.scene.rigidbody_world.steps_per_second
793.
794.            # Get property data
795.            defObjs = bpy.app.driver_namespace["force-contour"]
796.            defObjsDict = bpy.app.driver_namespace["force-contour-dict"]
797.
798.            # frame force dictionary
799.            forceDict = {}
800.            # for all constraints calculate object forces
801.            objects = bpy.data.objects
802.            for obj in objects:
803.                if(hasattr(obj.rigid_body_constraint, 'type')):
804.                    impulse = obj.rigid_body_constraint.appliedImpulse()
805.                    force = impulse / time_scale * steps_per_second
806.
807.                    if obj.rigid_body_constraint.object1.name in forceDict:
808.                        forceDict[obj.rigid_body_constraint.object1.name] += force
809.                    else:
810.                        forceDict[obj.rigid_body_constraint.object1.name] = force
811.
812.                    if obj.rigid_body_constraint.object2.name in forceDict:
813.                        forceDict[obj.rigid_body_constraint.object2.name] -= force
814.                    else:
815.                        forceDict[obj.rigid_body_constraint.object2.name] = 0 - force
816.
817.
818.
```

```
819.            if (not bpy.context.scene.explosion_tool.logfilepath):
820.                bpy.context.scene.explosion_tool.logfilepath = "forcelogfile.log"
821.            f = open(bpy.context.scene.explosion_tool.logfilepath, "a+")
822.            listForces = []
823.            for defObj in forceDict:
824.                f.write(str(defObj) + ":" + str(forceDict[defObj]) + "\n")
825.                listForces.append(forceDict[defObj])
826.
827.            minf = 0.0
828.            maxf = 0.0
829.
830.            for li in listForces:
831.                if li < minf:
832.                    minf = li
833.
834.                if li > maxf:
835.                    maxf = li
836.
837.
838.
839.            d = -1
840.            for defObj in forceDict:
841.                d += 1
842.                if d %100 == 0: sys.stdout.write('\r' +"%d" %d)
843.                objectd = bpy.data.objects[defObj]
844.                defObjForce = defObjs[defObjsDict[defObj]][1] # old force
845.
846.                # print("old force: ", defObjForce)
847.                # f.write("old force" + str(defObjForce) + ",")
848.
849.                ### Find material
850.                objForce = forceDict[defObj]   # new force
851.
852.                # print("new force: ", objForce)
853.                # f.write("new force: " + str(objForce) + ",")
854.
855.                # Calculate deformation differences
856.                defObjForceDif = defObjForce - objForce
857.
858.                defObjs[defObjsDict[defObj]][1] = objForce # set new force
859.                defObjs[defObjsDict[defObj]][2] = defObjForceDif # set new forceDif
860.
861.                # print("defObjForceDif: ", defObjForceDif)
862.                # f.write("defObjForceDif: " + str(defObjForceDif) + ",")
```

```
863.
864.
865.                dif = abs(defObjForceDif)
866.
867.                diff = dif /2000
868.
869.                if diff < displaySteps: step = int(diff)
870.                else: step = displaySteps -1
871.
872.
873.
874.                stepLast = objectd["defLastStep"]
875.                if step < stepLast -fadeDecrement: step = stepLast -fadeDecrement
876.
877.                mat = bpy.data.materials[materialName +"%03d" %step]
878.                objectd["defLastStep"] = step
879.
880.
881.
882.                objectd.material_slots[-1:][0].material = mat
883.
884.
885.
886.
887.        ###############################################################################
888.
889.        # Registration
890.        ###############################################################################
891.
892.        def register():
893.
894.            bpy.utils.register_class(SceneItems)
895.            bpy.utils.register_class(AmountItems)
896.            bpy.utils.register_class(ForceLogFileOperator)
897.            bpy.utils.register_class(DebrisLogFileOperator)
898.            bpy.utils.register_class(AddButtonOperator)
899.            bpy.utils.register_class(MassButtonOperator)
900.            bpy.utils.register_class(BuildButtonOperator)
901.            bpy.utils.register_class(ButtonOperator)
902.            bpy.utils.register_class(DeleteOperator)
903.            bpy.types.Scene.collection = bpy.props.CollectionProperty(type=SceneItems)
904.            bpy.types.Scene.amountcollection = bpy.props.CollectionProperty(type=AmountItems)
905.            bpy.utils.register_class(CustomDrawOperator2)
906.            bpy.utils.register_class(CustomDrawOperator)
```

```
            bpy.utils.register_class(ExplosionVariables)
            bpy.utils.register_class( ExplosionOperator )
            bpy.types.Scene.explosion_tool = bpy.props.PointerProperty(type=ExplosionVariables)
            bpy.utils.register_class(FancyPanel)
            bpy.utils.register_class(ExplosionPanel)
            bpy.utils.register_class(MessageBox)
            bpy.utils.register_class(RegisterHandlerOperator)
            bpy.utils.register_class(UnregisterHandlerOperator)
            bpy.utils.register_class(HandlerPanel)


        def unregister():
            del bpy.types.Scene.explosion_tool
            del bpy.types.Scene.collection
            del bpy.types.Scene.amountcollection
            bpy.utils.unregister_class(ExplosionVariables)
            bpy.utils.unregister_class(ExplosionPanel)
            bpy.utils.unregister_class( ExplosionOperator )
            bpy.utils.unregister_class(CustomDrawOperator2)
            bpy.utils.unregister_class(CustomDrawOperator)
            bpy.utils.unregister_class(SceneItems)
            bpy.utils.unregister_class(AmountItems)
            bpy.utils.unregister_class(ForceLogFileOperator)
            bpy.utils.unregister_class(DebrisLogFileOperator)
            bpy.utils.unregister_class(AddButtonOperator)
            bpy.utils.unregister_class(MassButtonOperator)
            bpy.utils.unregister_class(BuildButtonOperator)
            bpy.utils.unregister_class(ButtonOperator)
            bpy.utils.unregister_class(DeleteOperator)
            bpy.utils.unregister_class(FancyPanel)
            bpy.utils.unregister_class(MessageBox)
            bpy.utils.unregister_class(RegisterHandlerOperator)
            bpy.utils.unregister_class(UnregisterHandlerOperator)
            bpy.utils.unregister_class(HandlerPanel)
            # Clear property
            if "force-contour" in bpy.app.driver_namespace:
                del bpy.app.driver_namespace["force-contour"]
            if "force-contour-dict" in bpy.app.driver_namespace:
                del bpy.app.driver_namespace["force-contour-dict"]
            if "explosion" in bpy.app.driver_namespace:
                del bpy.app.driver_namespace["explosion"]
            if "start-frame" in bpy.app.driver_namespace:
                del bpy.app.driver_namespace["start-frame"]

```

```
951.
952.       if __name__ == "__main__":
953.           try: unregister()
954.           except: pass
955.           register()
```

Codes developed in MATLAB for producing the explosion contours are as follows:

Codes for variation of overpressure contours vs. time:

```
function overpressure_ref(a)
w=a^(1/3);
[z,t] = meshgrid (linspace(0,100),linspace(0,60));
p =  808 .*
(1+(z./(w*4.5)).^2)./((1+(z./(w*0.048)).^2).^0.5.*(1+(z./(w*0.32)).^2).^0.5.*(1+(z./(w*1.35)).^2).^0.5).*(1-t./(
0.001*w*980 .* (1+(z/(w*0.54)).^10)./((1+(z/(w*0.02)).^3).*(1+(z./(w*0.74)).^6).*(1+(z./(w*6.9)).^2).^0.5))).*exp((-
1)*t./(0.001*w*980 .* (1+(z/(w*0.54)).^10)./((1+(z/(w*0.02)).^3).*(1+(z./(w*0.74)).^6).*(1+(z./(w*6.9)).^2).^0.5)));
figure(3)
[c,h] = contour(t,p,z,1000);
%clabel(c,h,'FontSize',7,'Color','red');
%clabel(c,h);
title('Overpressure Contours');
xlabel('Time (s)');
ylabel('Overpressure Ratio p/pa');
end
```

Codes for explosion intensity contours:

```
function overpressure_scaledDistance_contours2()
[z,w] = meshgrid (linspace(0,50),linspace(0,100000));
p0 = 808 .*
(1+(z./(w.^(1/3)*4.5)).^2)./((1+(z./(w.^(1/3)*0.048)).^2).^0.5.*(1+(z./(w.^(1/3)*0.32)).^2).^0.5.*(1+(z./(w.^(1/3)*1.
35)).^2).^0.5);
 [c,h] = contour(z,w,p0,80);
clabel(c,h,'FontSize',8,'Color','red');
```

```
title('Overpressure Ratio Contours p0/pa');
xlabel('Scaled Distance (m)');
ylabel('W (kg s of TNT)');
end
```

Codes for overpressure ratio contours:

```
function overpressure_scaledDistance_contours2()
[z,w] = meshgrid (linspace(0,50),linspace(0,100000));
p0 = 808 .*
(1+(z./(w.^(1/3)*4.5)).^2)./((1+(z./(w.^(1/3)*0.048)).^2).^0.5.*(1+(z./(w.^(1/3)*0.32)).^2).^0.5.*(1+(z./(w.^(1/3)*1.
35)).^2).^0.5);

[c,h] = contour(z,w,p0,80);
clabel(c,h,'FontSize',8,'Color','red');
title('Overpressure Ratio Contours p0/pa');
xlabel('Scaled Distance (m)');
ylabel('W (kg s of TNT)');
end
```

Codes for variation of overpressure and total overpressure ratio vs scaled distance for a 2-span bridge exploded by 1 ton of explosives:

```
function overpressure_Plymoth_Bridge(a)
w=a^(1/3);
z = linspace(-20,20);
p0_mid = 808 .*
(1+(z./(w*4.5)).^2)./((1+(z./(w*0.048)).^2).^0.5.*(1+(z./(w*0.32)).^2).^0.5.*(1+(z./(w*1.35)).^2).^0.5);
p0_support1 = 808 .*
(1+((z+20)./(w*4.5)).^2)./((1+((z+20)./(w*0.048)).^2).^0.5.*(1+((z+20)./(w*0.32)).^2).^0.5.*(1+((z+20)./(w*1.35)).^2)
.^0.5);
p0_support2 = 808 .* (1+((z-20)./(w*4.5)).^2)./((1+((z-20)./(w*0.048)).^2).^0.5.*(1+((z-
20)./(w*0.32)).^2).^0.5.*(1+((z-20)./(w*1.35)).^2).^0.5);
plot(z,p0_mid,'r')
hold on
plot(z,p0_support1,'b')
hold on
```

```
plot(z,p0_support2,'g')
title('Overpressure Ratio vs Scaled Distance');
xlabel('Scaled Distance (m)');
ylabel('Overpressure Ratio p0/pa');
p0_total = p0_mid + p0_support1 + p0_support2;
figure(2);
plot(z,p0_total);
title('Total Overpressure Ratio vs Scaled Distance');
xlabel('Scaled Distance (m)');
ylabel('Total Overpressure Ratio p0/pa');
figure(3)
[z,t] = meshgrid (z,linspace(0,5));
p =  p0_total.*(1-t./( 0.001*w*980 .*
(1+(z/(w*0.54)).^10)./((1+(z/(w*0.02)).^3).*(1+(z./(w*0.74)).^6).*(1+(z./(w*6.9)).^2).^0.5))).*exp((-
1)*t./(0.001*w*980 .* (1+(z/(w*0.54)).^10)./((1+(z/(w*0.02)).^3).*(1+(z./(w*0.74)).^6).*(1+(z./(w*6.9)).^2).^0.5)));
contour(t,p,z,100);
```