AKADEMIA GÓRNICZO-HUTNICZA

IM. STANISŁAWA STASZICA W KRAKOWIE



Wydział Inżynierii Metali i Informatyki Przemysłowej



PRACA DYPLOMOWA MAGISTERSKA/INŻYNIERSKA

pt.

„Temat Pracy”

Imię i nazwisko dyplomanta: **Autor**

Kierunek studiów: **Informatyka stosowana**

Specjalność: **Systemy Informatyku Przemysłowej**

Nr albumu: …

Promotor: …

Recenzent: …

Podpis dyplomanta: Podpis promotora:

Kraków rok

„Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.”, a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.) „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej "sądem koleżeńskim"”, oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.”

Kraków, dnia ………….… Podpis dyplomanta …………….

# Spis Treści

[Spis Treści 3](#_Toc43481969)

[**1. Wstęp.** 4](#_Toc43481970)

[**2. Przegląd dostępnych postprocesorów.** 6](#_Toc43481971)

[**3. Opis aplikacji.** 7](#_Toc43481972)

[**4. Unreal Engine 4.** 10](#_Toc43481973)

[**5. Grafika niskopoziomowa zastosowana w aplikacji.** 11](#_Toc43481974)

[**6. Szczegółowy opis działania aplikacji.** 15](#_Toc43481975)

[**6.1 Korzystanie z postprocesora.** 15](#_Toc43481976)

[**6.2 Sterowanie.** 16](#_Toc43481977)

[**6.3 Wczytywanie modelu.** 18](#_Toc43481978)

[**7. Format pliku.** 19](#_Toc43481979)

[**8. Przykładowy eksporter dla programu Abaqus.** 19](#_Toc43481980)

[**9. Analiza wyników.** 19](#_Toc43481981)

[**10. Podsumowanie.** 19](#_Toc43481982)

[Bibliografia 20](#_Toc43481983)

[**Dodatek** 21](#_Toc43481984)

[**D1. Przykładowy format wejściowy i jego widok.** 21](#_Toc43481985)

[**D2. Przykładowe wejście do GUI.** 23](#_Toc43481986)

[**D3. Przykładowe zdjęcia z programu.** 23](#_Toc43481987)

**1. Wstęp.**

Żyjemy w czasach, w których ciężko jest sobie wyobrazić powstawanie nowych produktów bez symulacji numerycznych. Trudno jest sobie uzmysłowić zrobienie kroku wstecz i powrotu do przygotowywania nowych modeli przedmiotów codziennego użytku, czy też zaawansowanych maszynerii na papierze, przy pomocy ekierki i kalkulatora. Z pomocą przychodzą nam programy do wspomagania obliczeń inżynierskich takie jak Abaqus czy Ansys. Narzędzia te cechują się bardzo dobrą skuteczności w swojej pracy, lecz brakuje im pewnej swobody pokazywania wyników swoich operacji. Operator ma przeważnie możliwość obejrzeć policzony wyników bezpośrednio w programie, w którym pracuje lecz niejednokrotnie ma on problemy natury czysto technicznej z wyświetlanym obrazem. Postprocesor, bo o nim tutaj mowa, przeważnie nie jest skonstruowany jako główna część programu pakietu obliczeniowego, tylko jako kwestia drugorzędna umożliwiająca podejrzenie wartości w poszczególnych węzłach lub elementach. Inżynier musi obracać przy pomocy myszy model w taki sposób, aby ujrzeć interesujący go fragment modelu. Rozwiązanie jest to przeważnie niewygodne. Każdy obrót naszej kamery musi spotkać się z ponownym ustawianiem wyniku symulacji w pozycji wyśrodkowanej i odpowiednio przeskalowanej. A co gdyby skorzystać z pomocy narzędzia specjalnie stworzonego do wygodnego oglądania obiektów wyrenderowanych.

Odpowiedzią na to pytanie może być silnik graficzny służący do tworzenia gier komputerowych Unreal Engine 4 firmy Epic Games. Po mimo głównego zastosowania jakim jest oczywiście tworzenie programów służących do rozrywki, można go jak najbardziej zaadaptować do rozwiązań czysto inżynierskich takich jak utworzenie postprocesora dla wszystkich dostępnych programów wspomagających obliczenia inżynierskie. Możliwe jest to poprzez zastosowanie niezależnego formatu powstałego specjalnie dla tego programu oraz utworzenie postprocesora właśnie przy pomocy wspomnianego silnika graficznego. Rozwiązanie to łączy szybkość języka C++, który posłużył za konwerter plików tekstowych zawierających wszystkie niezbędne dane związane z modelem i sposobem jego renderowania oraz wyświetlaniem grafiki trójwymiarowej przy pomocy karty graficznej. Połączenie tych dwóch aspektów pozwoliło uzyskać postprocesor o bardzo krótkim czasie wczytywania się plików i oglądaniu wygenerowanych modeli z perspektywy „gracza” ze szczególnie prostym i intuicyjnym interfejsem użytkownika.

Elementem, bez którego niestety zaproponowany projekt nie może się obejść jest konieczność tworzenia eksporterów do każdego programu komercyjnego zajmującego się symulacjami numerycznymi. Wszystko ma jednakże swoje plusy i minusy. Zaprezentowany postprocesor jest uniwersalny, a format pliku importowanego jest wyjątkowo prosty, więc może posłużyć do renderowania modeli zarówno dwu jak i trójwymiarowych symulacji z dowolnego środowiska. Wyeksportowany w ten sposób model symulacji waży znacznie mniej co znacznie zmniejsza problematykę przenoszenia poszczególnych wyników symulacji. Dla przedstawionego programu nie ma również znaczenia czy wartości są przechowywane w węzłach czy w elementach. Automatycznie przełącza on swoje działanie i wyświetla wyniki ze specjalnie dobranej tekstury.

Specjalnie dla pokazania możliwości przedstawionego w tej pracy postprocesora została przygotowana wtyczka dla komercyjnego programu Abaqus firmy Simulia. Skrypt ten wyciąga niezbędne dane do przedstawienia zasymulowanego modelu już bezpośrednio w formacie docelowym dla postprocesora napisanego w silniku graficznym Unreal Engine 4.

W dalszych rozdziałach tej pracy zostanie szczegółowo opisane działanie obydwu wymienionych wyżej programów, ich możliwości oraz specyfikę działania. Przedstawione również zostaną wyniki porównujące kilka aspektów takich jak szybkość działania poszczególnych rozwiązań, waga plików wyeksportowanych oraz analiza porównawcza wizualnych efektów pracy.

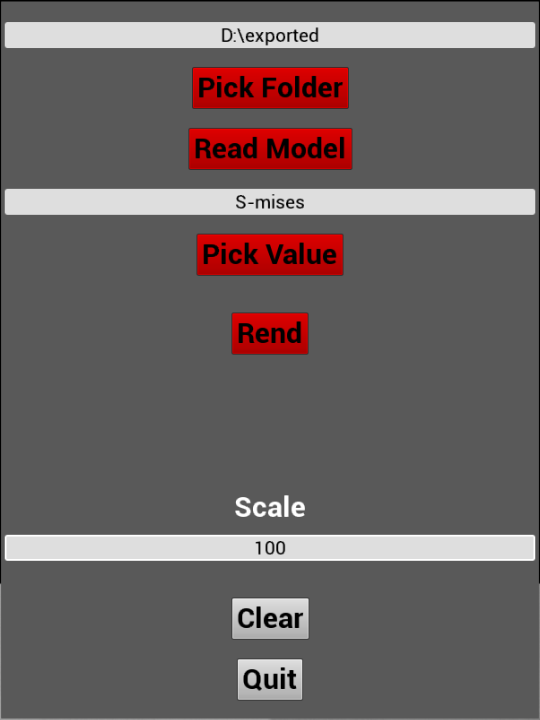
**2. Cel pracy.**

Silniki do tworzenia oprogramowania typu gry komputerowe, często nazywane po prostu „silniki gier” kojarzą się przede wszystkim z dostarczaniem programów służących czysto do celów rozrywkowych. Przekonanie to jest słuszne, lecz należy sobie postawić pytanie czy nadają się tylko i wyłącznie do tych celów? Otóż odpowiedź jest przecząca. Celem tej pracy było utworzenie postprocesora do aplikacji wspomagania inżynierów w ich codziennym projektowaniu oraz dążeniu do uzyskania satysfakcjonujących wyników. Przede wszystkim zadaniem postawionym przed tym oprogramowaniem jest otrzymanie jak najbardziej wiarygodnego odwzorowania modelu zasymulowanego w programach typu Abaqus czy Catia z możliwością obejrzenia go w sposób odchodzący od tradycyjnego. Drugim zadaniem jakie zostało postawione jest uzyskanie znacznie mniejszej wagi plików niezbędnych do wyrenderowania bazy wynikowej. Wszystko to zostało szczegółowo opisane w niniejszej pracy i ponadto została wykona szczegółowa analiza czasów wykonywania się operacji eksportu, importu oraz samego renderingu modelu zasymulowanego.

**3. Opis aplikacji.**

Główną ideą, która przyświecała stworzeniu tej aplikacji, było utworzenie uniwersalnego środowiska na wzór postprocesora aplikacji MES’owskich, gdzie użytkownik niezależnie od programu w jakim stworzył dany projekt, mógł zobaczyć wynik swoich działań w przyjaźniejszym formacie. Kolejnym istotnym elementem tej pracy jest zmniejszenie pamięci wymaganej, aby wyświetlić obliczony model oraz badanie możliwości szybkości renderowania grafiki trójwymiarowej w środowisku Unreal Engine 4.

Główną zaletą programu do wyświetlania wyników jest jego prostota. Od potencjalnego użytkownika wymagane jest tylko wskazanie ścieżki do folderu z wyeksportowanymi danymi do wyświetlenia i wybranie wartości jaką chciałby wyświetlić. Wygląd interfejsu użytkownika można zobaczyć zdjęciu Rys. 1. Warto wspomnieć, że aplikacja wyświetli wszystko co jest zgodne z przyjmowanym formatem plików. Renderowane obrazy nie muszą być plikami eksportowanymi z baz danych. Mogą to być pliki napisane własnoręcznie przez użytkownika, jeżeli istnieje takowa potrzeba.



Rys. 1 Interfejs użytkownika

Po określeniu wszystkich niezbędnych parametrów na ekranie zostaje wyświetlony model zadany przez użytkownika. Od teraz można swobodnie poruszać się wokół przedstawionego wyniku. Kamera posiada dwie możliwości sterowania. Za pomocą myszki zmienia się kąt patrzenia, natomiast za sprawą klawiatury kamera zmienia swoje fizyczne miejsce w przedstawionym świecie. Takie złożenie ruchu pozwala na pełne wykorzystanie możliwości zaprezentowanej aplikacji. Operator ma możliwość, aby rozglądać się dookoła w pełnym spektrum 360°, ale tylko na osi horyzontalnej. W przypadku osi wertykalnej istnieje ograniczenie tylko do 180° zastosowane specjalnie w celu uniknięciu możliwości zgubienia orientacji przez użytkownika. Przez użycie opisanego rozwiązania nie ma możliwości „przekręcenia” kamery w sposób niewygodny dla operatora.

Poruszanie widokiem jest natomiast w pełni nieskrępowane, aby można było dokładnie przyjrzeć się wyświetlonemu modelowi. Użytkownik ma możliwość na zbliżenie się bezpośrednio do interesującego go obiektu, bez możności wniknięcia w ten obiekt. Opcja ta została zablokowana specjalnie, aby uniknąć powstawania artefaktów związanych z renderowaniem w środowisku Unreal Engine 4. Istnieje jednak rozwiązanie problemu zaglądania wewnątrz zasymulowanego przedmiotu, ale wymagane jest specjalne wygenerowanie bazy wynikowej co zostanie opisane bliżej w rozdziale poświęconym wtyczce do programu Abaqus.

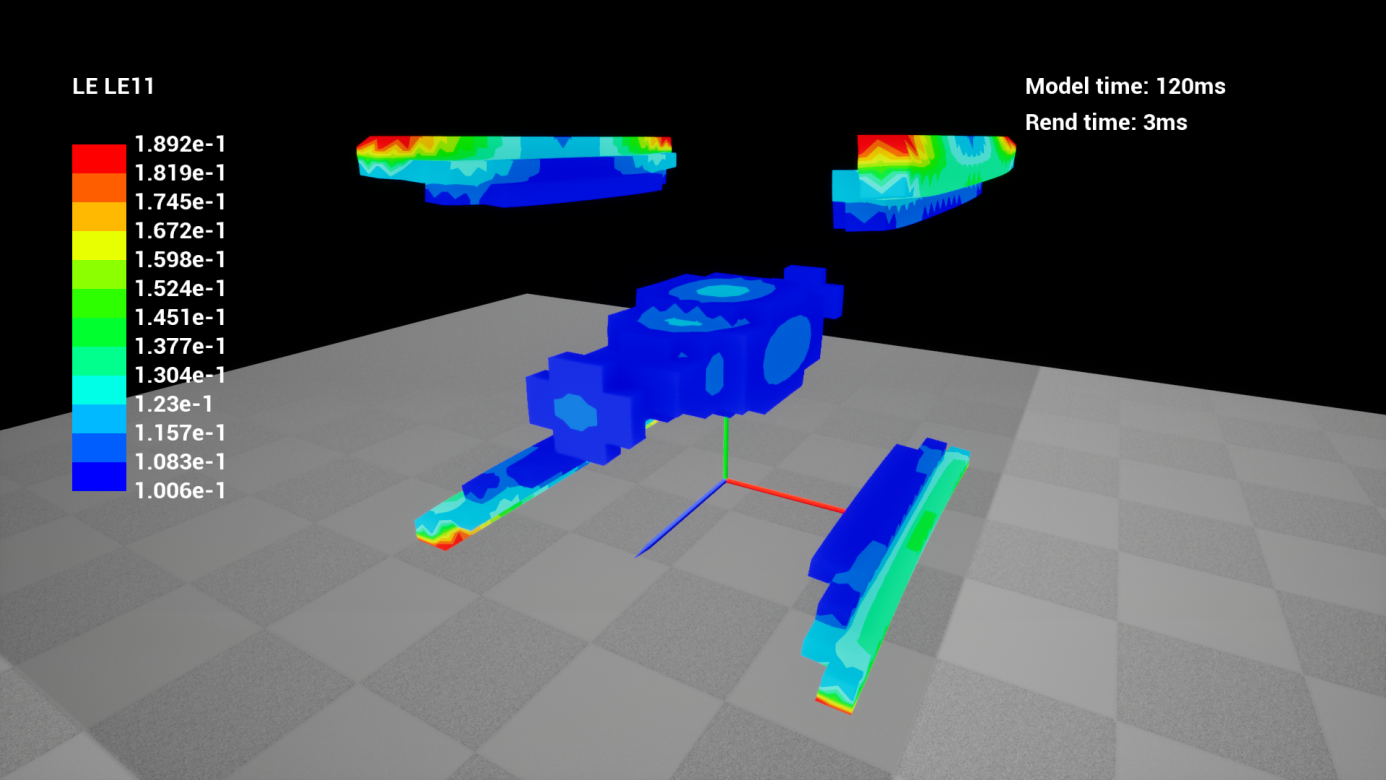
Renderowany model jest wyświetlany przy pomocy niskopoziomowej wtyczki do silnika Unreal Engine 4 służącej do generowania obrazu z zadanych trójkątów. Kolory natomiast zostają zmapowane na elementach ze specjalnie przygotowanej wcześniej tekstury o wysokiej jakości, będącą dyskretnym gradientem barw występujących w modelu odpowiadającym odpowiednim wartościom z bazy danych. Składowe te są brane z wyeksportowanego formatu i przypisywane odpowiednio elementom lub wierzchołkom w zależności od specyfiki danego typu wyniku. Tekstura jest mapowana w taki sposób, aby dokładnie przedstawić przejścia pomiędzy wartościami i jak najlepiej oddać gradient zmian.

Dodatkowo dla ułatwienia korzystania z przedstawionego rozwiązania zostało dodane kilka pomocnych udogodnień. Pierwszym z nich jest przełączanie się pomiędzy kamerą domyślną, którą steruje użytkownik, a kamerami ustawionymi w stałych orientacjach. Istnieje sześć takich kamer i są to odpowiednio widoki od góry, dołu z lewej oraz prawej strony, a także widok od przodu i tyłu. Kamery te nie maja możliwości zmiany kąta swojego „patrzenia” natomiast posiadają możliwość poruszania się w orientacji góra, dół, lewo, prawo, a także przybliżanie się i oddalanie się. Widoki te zapamiętują swoją orientację, więc po zmianie aktualnie wybranej kamery nie resetują się do początkowych ustawień. Pozwala to na dużą swobodę operacji przez użytkownika, czy też przygotowanie sobie widoków do ewentualnych zrzutów ekranu.

Kolejnym udogodnieniem jest przycisk resetu ułożenia kamery głównej. Ustawia on ponownie koordynaty widoku na startowe pozwalając na powrót w przypadku, gdy użytkownik z jakiegoś powodu nie będzie w stanie określić swojej pozycji, bądź też zablokuje się w wygenerowanym modelu. Utknięcie w wyrenderowanej grafice jest praktycznie niemożliwe, ale gdyby jednak nastąpiła taka sytuacja nie ma konieczności resetowania całej aplikacji wystarczy zresetować swoją pozycję. Współrzędne te są obliczane w momencie generowania obrazu zadanego przez operatora i specjalnie są przesunięte względem renderowanego modelu tak, aby uniemożliwić umieszczenie kamery początkowej w środku obiektu.

Ostatnia zaimplementowana funkcjonalność jest nie tyle udogodnieniem, ale niezbędnym elementem postprocesora, czyli pokazywanie legendy wartości. Po wciśnięciu odpowiedniego klawisza z boku ekranu wyświetla się grafika z zakresami wartości oraz kolorem jaki ten zakres przedstawia. Wartości przedstawione są w notacji naukowej z powodu bardzo dużych wahań pomiędzy poszczególnymi wartościami – cześć wyników posiada swoje wartości niezbyt przyjazne do celów porównawczych, więc by uniknąć kłopotów związanych z wyświetlaniem bardzo dużych, albo bardzo małych wyników została zastosowana właśnie taka notacja.

Dokładniejszy opis powyższych funkcjonalności wraz z przypisaniem do nich klawiszy zostanie opisane w rozdziale poświęconym szczegółowemu działaniu aplikacji. Poniżej na można zobaczyć przykładowy wygląd wyniku pracy postprocesora opisywanego w tej pracy. Szczególną uwagę przykuwa dokładność odwzorowania kolorów oraz ich nasycenie, co znacznie poprawia możliwości porównywania poszczególnych wartości.



Rys. 2 Przykładowy wygląd wyniku

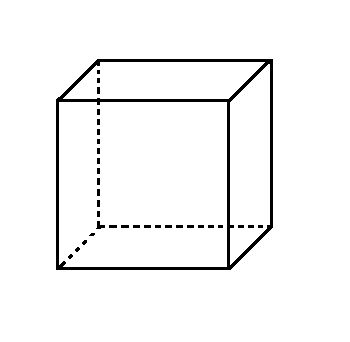
**4. Unreal Engine 4.**

**5. Grafika niskopoziomowa zastosowana w aplikacji.**

Główną inspiracją do powstania tej pracy było znalezienie odpowiedniej wtyczki pozwalającej na budowanie grafiki trójwymiarowej w zaawansowanym środowisku w sposób niskopoziomowy. Zaletą takiego sposobu wykonania jest na pewno możliwość wykorzystywania całego dostępnego interfejsu jaki gwarantuje silnik graficzny z możliwością wyrenderowania modelu, który de facto jeszcze nie istnieje. Korzystanie z technologii jaką jest Unreal Engine 4 zakłada używania modeli, które zostały poprzednio przygotowane w specjalnie do tego celu stworzonych programach. W klasycznym podejściu graficy najpierw tworzą wysoko zaawansowane modele graficzne składające się przeważnie z bardzo dużej ilości wierzchołków, połączonych krawędziami, a następnie tworzą z tych połączeń płaskie powierzchnie (bądź, też przy zastosowaniu odpowiednich sztuczek związanych z ustawieniem oświetlenia płaskie powierzchnie wyglądają na wykrzywione). Kolejnym krokiem jest utworzenie, dla każdego modelu osobnych animacji, tak aby postacie, rośliny, przedmioty mogły się „poruszać”.

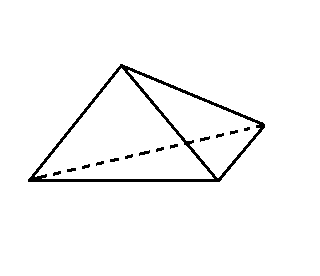
W taki sposób wygląda klasyczne podejście do wykorzystywania narzędzi programistycznych tego typu. Jednakże istnieje również możliwość renderowania modeli „w locie”. Otwiera to przed programistami wiele ciekawych wrót do popisu. Tworzenie programów w silnikach do gier nie ogranicza się teraz do utworzenia modeli w osobnym środowisku, a następnie zaimportowania ich trudów do silnika (przeważnie generuje to rozdzielenie twórców na grafików i programistów, co z kolei wiąże się z zatrudnianiem wykwalifikowanych pracowników w wąskim paśmie zdolności), lecz pozwala to na stworzenie klas (Unreal Engine 4 wykorzystuje do tworzenia własnej zawartości język C++ w sposób obiektowy) służących do wykonywania obliczeń oraz osobne do pokazywania wyników. Na potrzeby tego programu została utworzona jedna klasa, będąca jednocześnie importerem i jednostką generującą model niezbędny do renderowania grafiki trójwymiarowej. Większa ilość klas nie była niezbędna, z powodu bardzo dużego sprzężenia ze sobą procesu czytania danych wejściowych i umieszczania ich w specjalnych kontenerach wymaganych do poprawnego działania wykorzystanej wtyczki.

Generowanie modeli graficznych na podstawie importowanych plików nie różni się zbytnio od klasycznego podejścia. Każdy obiekt grafiki trójwymiarowej niezależnie od tego jak bardzo jest zaawansowany sprowadzany jest w dużym przybliżeniu do zbioru wierzchołków i trójkątów formujących powierzchnie. Jest to największa zaleta przyjęta jako główna zasada towarzysząca tworzeniu tego oprogramowania jakim jest własny importer, czyli: wszystko da się przedstawić przy pomocy odpowiednio pomalowanych trójkątów. Aksjomat ten można łatwo przybliżyć poprzez wyobrażenie sobie początkowo najprostszego typu elementu (element został ten zaimplementowany jako pierwszy, stąd też najprostszy) jako sześcianu, czyli elementu ośmiowęzłowego. Punkty całkowania umieszczone są wtedy w wierzchołkach. Każdy taki prostopadłościan posiada sześć ścian natomiast, każda taka ściana składa się z dwóch trójkątów prostokątnych równoramiennych o bokach długości a, a oraz . Wygląd takiego sześcianu został przedstawiony na Rys. 3.



Rys. 3 Element ośmiowęzłowy

W najprostszym przypadku wyrenderowanie takiej figury, będzie sprowadzało się do utworzenia kontenera przechowującego informacje o pozycji w świecie renderowanym poszczególnych trójkątów. Sposób ten jednak jest wysoce niewydajny z powodu bardzo dużego zużycia pamięci RAM karty graficznej. Musi ona wtedy zapamiętać pozycję każdego trójkąta, których ilość przeważnie sięga milionów w renderowanej scenie. Należy również pamiętać o fakcie występowania poszczególnych wierzchołków w kilku różnych trójkątach, co z kolei wiąże się z dużą redundancją danych. Dużo lepszym rozwiązaniem jest przechowywanie informacji z koordynatami wierzchołków w osobnym kontenerze i odwoływanie się do nich przy pomocy indeksów struktury, w której przechowujemy dane. Teraz powstawanie nowych trójkątów odbywa się w sposób znacznie bardziej zoptymalizowany. Każda nowa powierzchnia składa się z trójek odnośników w postaci indeksów do zapamiętanych koordynat zamiast samych redundantnych pozycji danych wierzchołków. W zależności od implementacji wersja redundantna zajmuje znacznie więcej pamięci od wersji optymalnej. Wszystko zależy od renderowanej figury, a przede wszystkim stosunku wierzchołków do trójkątów. Jeżeli w modelu jest znaczenie więcej powierzchni płaskich od punktów je tworzących to zysk jest większy. Łatwo jest to zrozumieć na przykładzie. Przedstawiona figura to czworościan foremny składający się czterech trójkątnych boków i czterech wierzchołków.

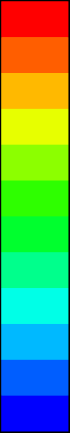


Rys. 4 Czworościan

Sposób nieefektywny daje nam cztery ściany, każda z nich posiada trzy wierzchołki, a każdy z wierzchołków ma trzy wartości składowe swojego fizycznego położenia w renderowanej scenie określane jako X, Y oraz Z. Dodatkowo składowe te są przechowywane w postaci zmiennych typu Float, ważącego 4 bajty. Ostatecznie daje nam to 144 bajty niezbędne do przechowywania naszego czworościanu. Natomiast wersja optymalna wymaga tylko zapamiętania położenia czterech wierzchołków (każdy z nich również posiada trzy składowe X, Y, Z przechowywane w zmiennej typu Float), a następnie dla każdej ściany zapamiętania indeksów poszczególnych wierzchołków. Dla rozpatrywanego ostrosłupa daje nam to zaledwie 60 bajtów. Jest to zaledwie 42% wagi rozwiązania nieoptymalnego, a stosunek ten maleje przeważnie z dodaniem, każdego nowego wierzchołka wraz z nowymi ścianami. Dodatkowo lepsze rozwiązanie pozwala nam łatwe policzenie wagi, każdego modelu.

Wracając do naszego najprostszego elementu jakim jest sześcian i stosując powyższe metody optymalizacyjne, wystarczy teraz zapamiętać tylko osiem wierzchołków i dwanaście ścian tworzących ten element. Dla ułatwienia całego procesu tworzenia i przejrzystości rozwiązania oraz poniekąd formatu jaki został narzucony z zaproponowanego eksportera z Abaqus’a, dla każdego typu elementu zarówno dwuwymiarowego jak i trójwymiarowego powstały funkcje tworzące ściany elementów w sposób automatyczny, przyjmujące jako argumenty indeksy wierzchołków, z których składa się element. Metody te generują trójki indeksów tworzących powierzchnie w odpowiedniej orientacji. Dla silnika Unreal Engine 4 ma bardzo duże znaczenie, czy dany model jest z „lewej” czy z „prawej” strony, gdyż renderuje obraz tylko z jednej. Oznacza to przede wszystkim, iż w przypadku pomyłki sześcian byłby „odwrócony do środka”. Z zewnątrz byłby przezroczysty natomiast od wewnątrz miałby normalnie widoczne ściany. Powoduje to bardzo duże problemy z poprawnym wyświetlaniem modelu, dlatego też powstały właśnie wcześniej wspomniane funkcje – w celu uniknięcia artefaktów graficznych.

Idąc dalej możemy zmienić postrzeganie naszego elementu z sześcianu na dowolną bryłę będącą sześcianoidem, czyli figurą trójwymiarowa posiadającą osiem wierzchołków i sześć ścian, ale o różnych długościach boków. Dodając kolejne sześcianoidy powiększamy tylko kontener przechowujący informacje o położeniu wierzchołków i dodajemy kolejny element przy pomocy funkcji do tego służącej przez co powiększamy listę trójkątów tworzących nasz model. W taki właśnie iteracyjny sposób w przybliżeniu buduje się model renderowany w postprocesorze opisywanym w tej pracy. Pozostaje jeszcze tylko jedno pytanie jak przedstawić wartości przechowywane w elementach, bądź punktach na ekranie? Najprostszą metodą, było by wy korzystanie sposobu opisywanego przez standard OpenGL, czyli ustawienie kolorów odpowiadających wartościom dla każdego z wierzchołków, a następnie pozwolić karcie graficznej policzyć gradient pomiędzy krańcami trójkąta w sposób automatyczny i w ten sposób uzyskać przejścia dla wszystkich wartości. Rozwiązanie to jednak nie sprawdziło się z dwóch głównych powodów. Po pierwsze uzyskany gradient wartości jest ciągły i ciężko jest uzyskać z niego przedział wartości w celu porównywania wyników. Znacznie lepiej w takiej sytuacji sprawuje się dyskretny gradient przedstawiony na . Drugim powodem jest brak wsparcia ze strony wykorzystywanej wtyczki do silnika graficznego. Najlepszy rozwiązaniem tego problemu jest wykorzystanie możliwości oteksturowania modeli, czyli nałożenie tekstury na powierzchnie płaskie. Tekstura jest niczym innym jak mapą bitową posiadającą kolory, które na jednym z etapów renderowania strumienia graficznego są nakładane na obiekt. Odbywa się to za sprawą wycinania fragmentów obrazu w koordynatach przechowywanych w osobnym kontenerze. Struktura ta jest dwuwymiarowa i przeważnie odpowiada indeksom jeden do jeden to znaczy dla wierzchołka o indeksie 248, punkt na teksturze przechowywany jest w kontenerze z koordynatami tekstury w miejscu o indeksie 248. Przy pomocy tego mechanizmu możliwe jest dokładne wskazywanie wartości na modelu.



Rys. 5 Tekstura

Dzięki zastosowaniu takich metod niskopoziomowych, było możliwe stworzenie wysokopoziomowych programów takich jak przedstawiony postprocesor. Opisane powyżej składowe zostały zaadoptowane i przekształcone w odpowiedni sposób, aby utworzyć narzędzie jak najbardziej przydatne dla inżynierów.

Bardzo dużą rolę w tej aplikacji bez wątpienia odgrywa wtyczka do silnika odpowiadająca za tworzenie grafiki. Przez cały czas jest ona nazywa właśnie w ten sposób z powodu tego, iż nie jest ona integralną częścią silnika. Została ona napisana przez twórców i dołączona do Unreal Engine 4, ale nie jako jedna ze składowych całości tylko jako dodatek, który może się okazać, że zostanie w przyszłości usunięty z powodu dużych braków w swoim działaniu. Po mimo wszystko okazała się bardzo przydatna do tworzenia tego oprogramowania.

**6. Szczegółowy opis działania aplikacji.**

**6.1 Korzystanie z postprocesora.**

Głównym atutem utworzonego oprogramowania jest wyjątkowa prostota jego interfejsu graficznego. W przeciwieństwie do komercyjnych programów nie ma potrzeby uczenia się skomplikowanych skrótów czy szukania odpowiednich ikonek odpowiadających za przesuwanie, skalowanie lub rotację obrazu. W tym podejściu model jest nieruchomy co również sprzyja wygodzie oglądania go. Nie ma potrzeby zastanawiania się w jakiej orientacji znajduje się obiekt rozpatrywany ponieważ zawsze jest on we właściwej przyjętej przestrzeni osi. Jednak za nim na ekranie wyrenderuje się jakikolwiek obraz należy wskazać interesujący nas model.

Sposób wyboru bazy danych jest nieskomplikowany. W pierwszym dostępnym polu należy wpisać ścieżkę dostępu do katalogu wyeksportowanego, gdzie znajduje się folder „model”. Jest to najważniejszy folder ponieważ zawiera on wszystkie niezbędne informacje potrzebne do wyrenderowania importowanego modelu. Pozostałe foldery zawierają tylko i wyłącznie informację na jaki kolor pomalować poszczególne części modelu. Bez tego folderu nic nie zostanie wyświetlone na ekranie.

Po wybraniu odpowiedniego folderu należy potwierdzić go wciskając przycisk „Pick Folder”. Od teraz program wie dokładnie, gdzie należy szukać informacji o bazie danych, która ma zostać pokazana. Kolejnym krokiem jest wciśnięcie przycisku „Read Model”. Te dwa procesy zostały rozdzielone, aby uniknąć problemów z podaniem niewłaściwej ścieżki dostępu. Dodatkowo przed wciśnięciem tego przycisku można ustalić skalę wyświetlanego modelu. Może się zdarzyć, że inżynier miał za zadanie utworzenie jakiegoś bardzo małego, bądź wręcz przeciwnie bardzo dużego modelu, natomiast program opisywany w tej pracy wyświetla wszystko zgodnie z jego rzeczywistymi wymiarami podanymi na etapie tworzenia symulacji np. w Abaqus’ie. Oznacza to, że kostka mająca boki o długości 2cm, będzie wyświetlana jako bardzo mały obiekt. Po to właśnie zostało dodane skalowanie, aby użytkownik mógł dokładnie powiększyć, bądź pomniejszyć obraz bez konieczności ponownego wykonywania symulacji, która mogła by być czasochłonna. Sytuacji takich należy unikać.

Każdorazowe wciśnięcie przycisku „Read Model” w przypadku, gdy już jakiś model był wczytany resetuje listę wierzchołków i wstawia do niej nowy model. Pozwala to na swobodne dostosowywanie wielkości renderowanego obiektu. Skalę ustala się wpisując jej wartość pod napisem „Scale”. Wielkość wpisywana w tym miejscu oznacza procentową skalę względem obiektu macierzystego. Domyślna wartość tego pola to 100 co oznacza brak zmian w wielkości, natomiast po wpisania 10 otrzymamy model pomniejszony 10 razy, dla 1000 model powiększony 10 razy itd.

Ostatnim krokiem jest wybranie wartości jaka ma posłużyć za nałożenie kolorów na obiekt i w ten sposób pokazać rozkład tej wartości w symulacji. Odbywa się to poprzez wpisanie w pole nad przyciskiem „Pick Value” folderu znajdującego się w podanej ścieżce dostępu oraz wartości znajdującej się w tym folderze w pliku \*.txt. Przykładowo, aby uzyskać widok naprężeni mises’a należy w tym polu wpisać „S-mises”. Szczegółowy opis formatu modelu wyeksportowanego znajduje się w osobnym rozdziale. Wartości we wszystkich polach podajemy bez żadnych znaków dodatkowych takich jak cudzysłów, czy apostrof. Na sam koniec wystarczy wcisnąć przycisk „Rend” i na ekranie powinien wyświetlić się oczekiwany model.

Jako jeden z mechanizmów zabezpieczających oraz ułatwiających pracę został zaimplementowany system zmiany kolorów przycisków. Włączniki odpowiadające za wybieranie odpowiednich opcji maja barwę czerwoną oraz zieloną. Kolor czerwony oznacza odpowiednio brak zatwierdzenia odpowiadającej mu kwestii lub brak możności wciśnięcia danego przycisku. Przykładowo jeżeli guzik „Pick Path” jest czerwony oznacza to, że żadna ścieżka nie została do tej pory wybrana. Dodatkowo należy pamiętać o hierarchii wyborów. Wszystko powinno zostać określone od góry do dołu. Oznacza to brak możliwości wybrania wartości przed wyborem modelu. Jest to podyktowane specyfiką ustalania wszystkich parametrów i wartości niezbędnych do wyrenderowania sceny. Ostatnim udogodnieniem jest raczej niezbędny system sprawdzania, czy rzeczywiście we wskazanej ścieżce znajduje się jakikolwiek model, bądź czy istnieje wartość taka jaką wskazaliśmy na etapie wybierania parametrów do wyświetlenia. Jeżeli cokolwiek się nie zgadza i postprocesor nie może odnaleźć wskazanych plików wyświetla użytkownikowi błąd w postaci powiadomienia.

W opisywanym interfejsie użytkownika znajdują się jeszcze dwa przyciski na samym dole okna. Są to odpowiednio „Clean” oraz „Quit”. „Clean” służy do czyszczenia obecnie wybranego modelu, czyli zresetowania wszystkich list przechowujących wartości oraz usunięcia poprzednio wyrenderowanego obiektu. Drugi z wymienionych przycisków zamyka aplikację.

**6.2 Sterowanie.**

Intuicyjne sterowanie to podstawa dobrego postprocesora. Użytkownik nie powinien tracić czasu na szukanie w obszernych dokumentacjach technicznych jak korzystać oprogramowania mającego za zadanie tylko i wyłącznie wyświetlać symulowane zjawisko. Takie rozwiązania ma swoje wady i zalety. Za plus na pewno można policzyć niski próg wejścia dla danej aplikacji. Inżynier nie będzie potrzebował więcej niż pięć minut, aby zaznajomić się ze wszystkimi możliwościami danego oprogramowania. Minusem na pewno jest brak wyszukanych użyteczności takich jak wyodrębnianie poszczególnych części modelu czy jakieś zaawansowane metody wyświetlania.

W niniejszym postprocesorze postawiono na szybkość wyświetlania i zmniejszenie wagi plików niezbędnych do wyświetlenia elementu, a co za tym idzie sterowania również jest wyjątkowo proste i intuicyjne. W tabeli zostały spisane i wytłumaczone wszystkie przyciski wykorzystywane w niniejszej aplikacji.

Tabela 1 Sterowanie

|  |  |
| --- | --- |
| **Klawisz** | **Zastosowanie** |
| Escape | Menu opcji – Interfejs Użytkownika |
| W | Ruch do przodu |
| A | Ruch w lewo |
| S | Ruch do tyłu |
| D | Ruch w prawo |
| Mysz | Obrót kamery |
| Z, V | Menu wartości |
| R | Reset kamery do miejsca początkowego |
| I | Informacje o czasie renderowania |
| 9 | Przełączenie kamery na widok od przodu |
| 7 | Przełączenie kamery na widok od tyłu |
| 8 | Przełączenie kamery na widok od góry |
| 2 | Przełączenie kamery na widok od dołu |
| 4 | Przełączenie kamery na widok z lewej |
| 6 | Przełączenie kamery na widok z prawej |
| 5 | Powrót do kamery ruchomej |
| Strzałka w lewo | Ruch kamery statycznej w lewo |
| Strzałka w prawo | Ruch kamery statycznej w prawo |
| Strzałka do góry | Ruch kamery statycznej do góry |
| Strzałka w dół | Ruch kamery statycznej w dół |
| + | Przybliżanie kamery statycznej |
| - | Oddalanie kamery statycznej |

W powyższej tabeli () zostały spisane i wytłumaczone w dużym skrócie wszystkie klawisze występujące w niniejszym postprocesorze. Główna część sterowania opiera się na wykorzystaniu klawiszy sterujących „WASD” oraz poruszania myszką. Wciskając odpowiedni przycisk kamera przesuwa się względem strony, w którą aktualnie jest zwrócona, a więc ruch do przodu będzie przesuwał kamerę pod kątem 90° ekranu. Będzie się on przybliżał do miejsca, gdzie obecnie patrzy użytkownik. Ruch do tyłu natomiast będzie działał odwrotnie, obraz będzie się oddalał. Ruch w lewo i prawo przesuwa nas odpowiednio na boki.

Do wykorzystania na potrzeby operatora zostało dodane również sześć kamer statycznych, które nie mają możliwości zmiany kąta patrzenie. Mają za to możliwość przesuwania się oraz przybliżania i oddalania. Ich głównym zadaniem jest pozostawania prostopadle do osi, na którą patrzą. Są czymś na wzór rzutów izometrycznych, bez zmiany renderowania modelu.

Użytkownik ma dodatkowo do swojej dyspozycji dwa widoki przedstawiające pewne wartości. Są to interfejs pokazujący wartości w postaci legendy oraz czasy wczytywania i renderowania wartości. Drugi z dostępnych interfejsów służy głównie celom diagnostycznym.

**6.3 Wczytywanie modelu.**

Cały proces utworzenia grafiki trójwymiarowej przy pomocy używanej wtyczki opiera się na wypełnieniu przynajmniej trzech list odpowiednich typów. Na podstawie tych kontenerów budowana jest cała grafika dotycząca renderowanej bazy danych. Nie dotyczy to już istniejącego „świata”, ani interfejsów użytkownika. Trzy wspomniane wcześniej listy to tabele wierzchołków, trójkątów oraz koordynat tekstur wierzchołków.

Najpierw zostanie opisana pierwsza wspomniana lista. Służy ona za magazyn wszystkich współrzędnych X, Y oraz Z każdego istniejącego wierzchołka występującego w importowanej symulacji. De facto jest to lista przechowująca wektor z trzema wartościami. Każdy indeks listy odpowiada bezpośrednio numerowi wierzchołka reprezentującego swoje położenie globalne w świecie.

Druga z wymienionych struktur danych przechowuje kolejne indeksy wierzchołków służące do budowania trójkątów wyświetlanych na ekranie. Jest to zwyczajna lista przechowująca wartości typu całkowitoliczbowego czyli Integer. W swoim założeniu jej liczba elementów zawsze powinna być podzielna przez 3. Jest to wynik podstawowego założenia o budowie kolejnych płaszczyzn. Wszystkie renderowane obiekty składają się z trójkątów, a więc muszą zostać dostarczone dokładnie 3 wierzchołki, aby utworzyć taką figurę płaską. Algorytm renderingu pobiera trójkami numery indeksów przechowywane w tej liście, a następnie sprawdza w poprzednio opisywanej strukturze pamięci, gdzie dokładnie znajdują się pobierane punkty i następnie pomiędzy nimi ukazuje się powierzchnia płaska.

Jednakże ta powierzchnia nie miałaby żadnego koloru, gdyby nie fakt istnienia trzeciej listy będącej wektorem dwu wartościowym, przechowującym współrzędne wierzchołka umieszczone na dwuwymiarowej teksturze. Przeważnie, każdy model posiada własną teksturę. Jest ona bitmapą posiadającą zamierzone kolory, które grafik chciałby przedstawić na swoim wygenerowanym modelu. W teorii grafiki przyjęło się umieszczanie tekstury w układzie kartezjańskim z początkiem w punkcie (0,0) w lewym dolnym rogu i końcu w punkcie (1,1) czyli prawym górnym rogu obrazka. W przypadku wykorzystywanej wtyczki współrzędne zostały odwrócone, gdyż punkt (0,0) znajduje się w lewym górnym rogu, a punkt (1,1) w prawym dolnym rogu tekstury. Po mimo tych drobnych zmian wszystkie mechanizmy teksturowania działają zgodnie z zamierzeniami.

Ostatnia lista zawiera właśnie współrzędne tekstury wierzchołków odpowiadających indeksom w tej tablicy. To znaczy, że w pod numerem 14 znajdują się współrzędne niezbędne do oteksturowania wierzchołka numer 14 (pamiętając oczywiście, że numerację indeksów zaczynamy od 0). Dzięki tej liście nasz model będzie posiadał kolory inne niż biały, który jest domyślną barwą dla wtyczki w trakcie renderowania modelu.

**7. Format pliku.**

**8. Przykładowy eksporter dla programu Abaqus.**

**9. Analiza wyników.**

**10. Podsumowanie.**

Wszystkie założenia jakie zostały postawione za cel tej pracy zostały zrealizowane. Zadnie jakim było wykorzystanie silnika stworzonego czysto do tworzenia gier i uzyskania przy jego pomocy narzędzia przydatnego inżynierom powiodło się z dużym pozytywnym skutkiem. Postprocesor wykonany w środowisku Unreal Engine 4 jest intuicyjny oraz szybki w swoim działaniu. Dodatkowo zysk w postaci utworzenia oprogramowania o wolnym formacie dostępnym dla wszystkich komercyjnych postprocesorów oraz zmniejszeniu wagi plików niezbędnych do eksportu jest ogromny. Na ostatnich stronach tej pracy znajduje się „Dodatek” w postaci kilku przykładów zarówno formatu jak i korzystania z samego postprocesora. Dziękuję za przeczytanie niniejszego dokumentu. (nie jestem pewien czy na końcu powinno znajdować się coś w stylu podziękowań czy też nie)

# Bibliografia

Anon., 2018. *The Python Tutorial — Python 2.7.15 documentation.* [Online]   
Available at: https://docs.python.org/2/tutorial/

Epic Games, 2020. *The most powerful real-time 3D creation platform - Unreal Engine.* [Online]   
Available at: https://www.unrealengine.com/en-US/  
[Data uzyskania dostępu: 19 Czerwiec 2020].

Epic Games, 2020. *Unreal Engine 4 Documentation | Unreal Engine Documentation.* [Online]   
Available at: https://docs.unrealengine.com/en-US/index.html  
[Data uzyskania dostępu: 19 Czerwiec 2020].

Puri, G., 2015. *Python Scripts for Abaqus Learn by Example.* Wydanie pierwsze red. Charleston: Gautam Puri, 2011.

**Dodatek**

**D1. Przykładowy format wejściowy i jego widok.**

Folder\_docelowy\model\COORD.txt

0.0 0.0 0.0

0.1 0.0 0.0

0.0 0.1 0.0

0.0 0.0 0.1

0.1 0.1 0.0

0.0 0.1 0.1

0.1 0.0 0.1

0.1 0.1 0.1

-0.1 0.2 -0.1

-0.1 0.2 0.2

0.2 0.2 0.2

0.2 0.2 -0.1

Folder\_docelowy\model\ELEMENTS.txt

5 3 0 2 7 6 1 4

9 5 2 8 10 7 4 11

Folder\_docelowy\model\scale\_factor.txt

1.0

Folder\_docelowy\value\value.txt

1

2

3

4

5

6

7

8

9

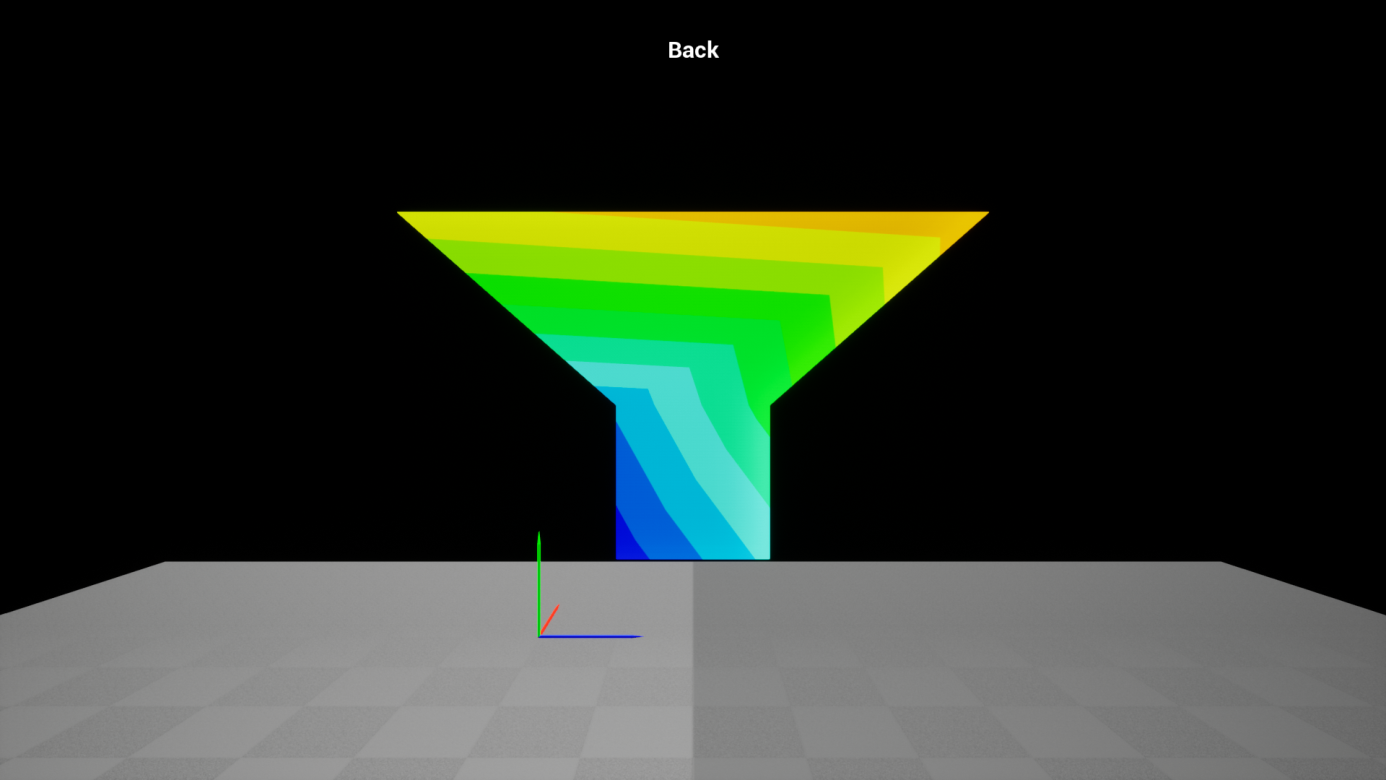
10

11

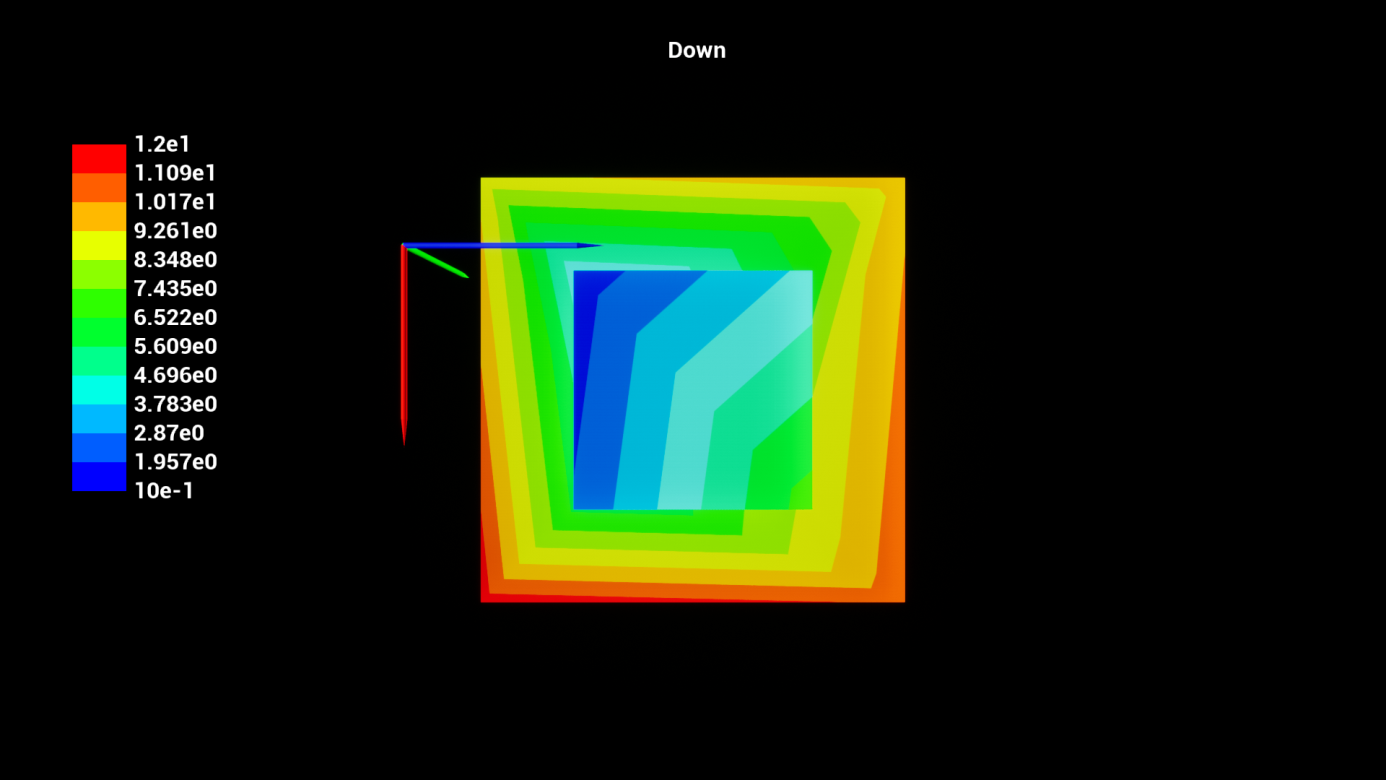
12

NODAL

Przedstawiony powyżej przykład plików jest pokazaniem modelu składającego się z dwóch elementów ośmiowęzłowych o wartościach nałożonych w węzłach. W tym modelu nie zostało uwzględnione przemieszczenie. Wyniki zostały pokazane na oraz .

****

Rys. 6 Widok przykładowego modelu z tyłu.

****

Rys. 7 Widok przykładowego modelu od dołu.

**D2. Przykładowe wejście do GUI.**

Poniżej na znajduje się przykładowo uzupełnione Menu postprocesora z następującymi parametrami:

- ścieżka dostępu – „E:\exported\3N\39”,

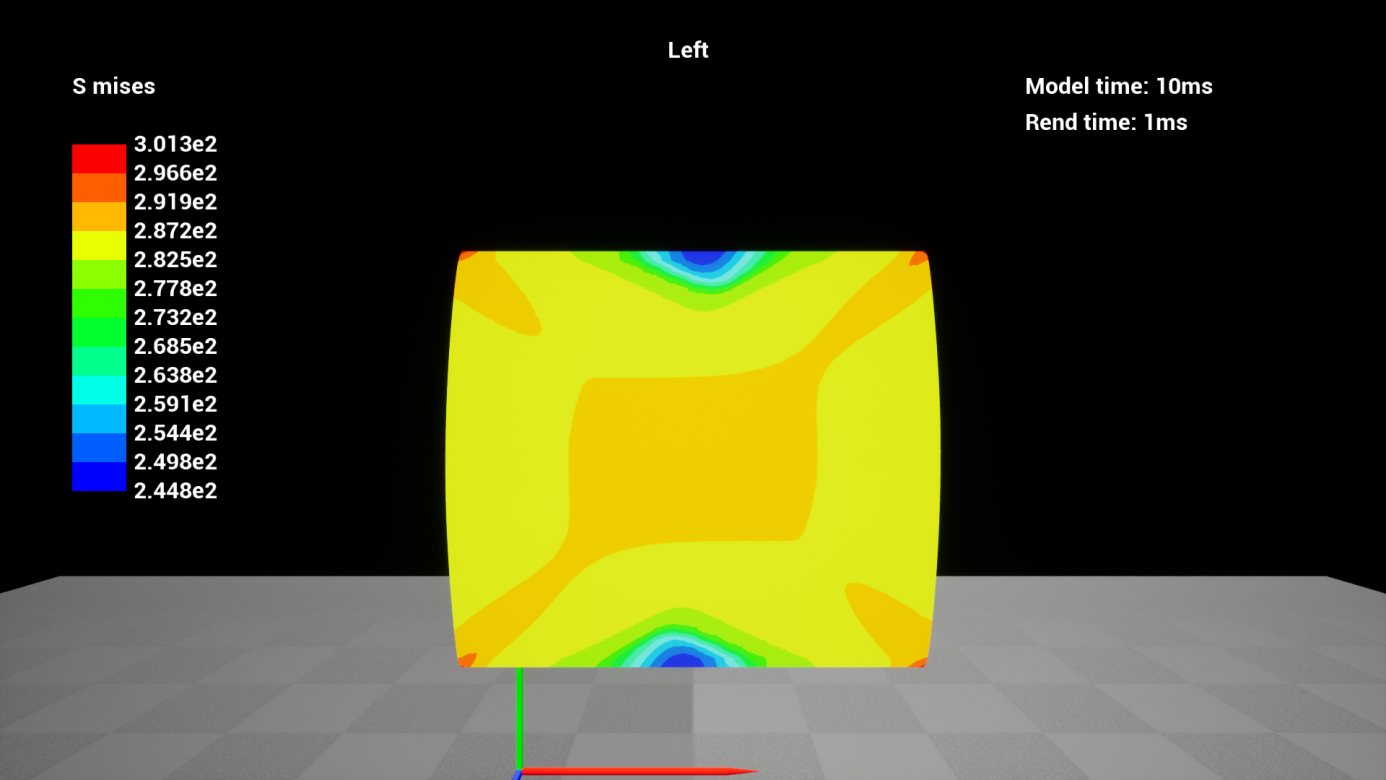
- wybrane wartości to „mises” z folderu „S”,

- skala ustawiona na 10% wielkości oryginału.

Jak widać wszystkie przyciski są koloru zielonego co oznacza poprawnie wczytany model i wszystkie jego wartości. Na rysunku widać wynik tych operacji, czyli dwuwymiarowy wynik (trójwęzłowy) wraz z legendą wartości oraz czasami importu i renderingu.

****

Rys. 8 Przykładowo wypełnione pola w Menu

****

Rys. 9 Widok uzyskany z parametrów podanych w przykładzie

**D3. Przykładowe zdjęcia z programu.**