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](#_Toc43129285)

[**1. Wstęp.** 4](#_Toc43129286)

[**2. Przegląd dostępnych postprocesorów.** 6](#_Toc43129287)

[**3. Opis aplikacji.** 7](#_Toc43129288)

[**4. Unreal Engine 4.** 9](#_Toc43129289)

[**5. Grafika niskopoziomowa zastosowana w aplikacji.** 10](#_Toc43129290)

[**6. Szczegółowy opis działania aplikacji.** 10](#_Toc43129291)

[**7. Format pliku.** 10](#_Toc43129292)

[**8. Przykładowy eksporter dla programu Abaqus.** 10](#_Toc43129293)

[**9. Analiza wyników.** 10](#_Toc43129294)

[**10. Podsumowanie.** 10](#_Toc43129295)

[**Bibliografia.** 10](#_Toc43129296)

[**Dodatek** 11](#_Toc43129297)

[**D1. Przykładowy format wejściowy i jego widok.** 11](#_Toc43129298)

[**Przykładowe zdjęcia z programu.** 13](#_Toc43129299)

**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. Przegląd dostępnych postprocesorów.**

**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. 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.

Zdjęcie z GUI

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.

Szczegółowy opis powyższych funkcjonalności wraz z przypisaniem do nich klawiszy zostanie opisane w rozdziale poświęconym szczegółowemu działaniu aplikacji.

Zdjęcie jakiegoś 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 WSTAW ODNOŚNIK DO RYSUNKU.

Obrazek z tym sześcianem.

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 WSTAW ODNOŚNIK DO OBRAZKA to czworościan foremny składający się czterech trójkątnych boków i czterech wierzchołków.

Obrazek czworościanu

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 WSTAW ODNOŚNIK DO RYSUNKU. 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.

Zdjecie tekstury.

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.

**6. Szczegółowy opis działania aplikacji.**

**7. Format pliku.**

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

**9. Analiza wyników.**

**10. Podsumowanie.**

**Bibliografia.**

**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

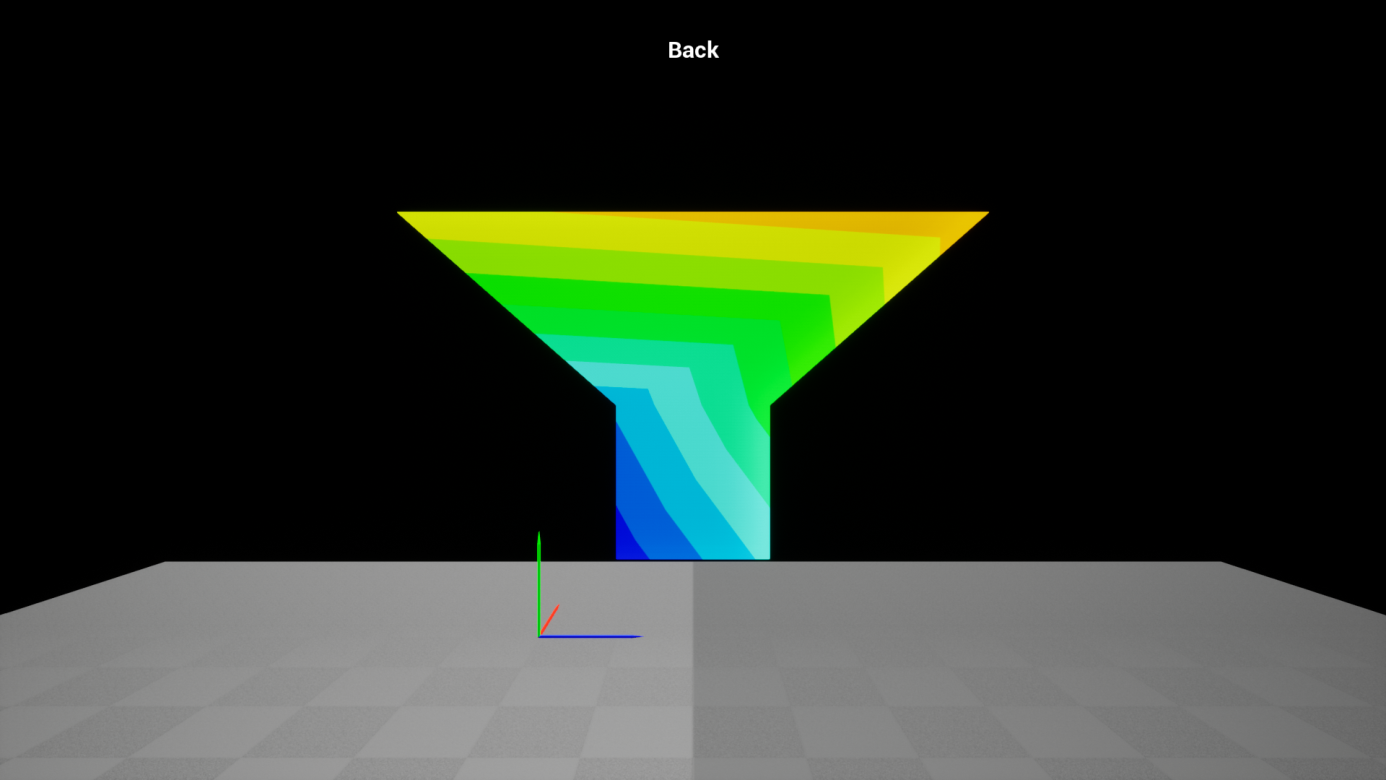
10

11

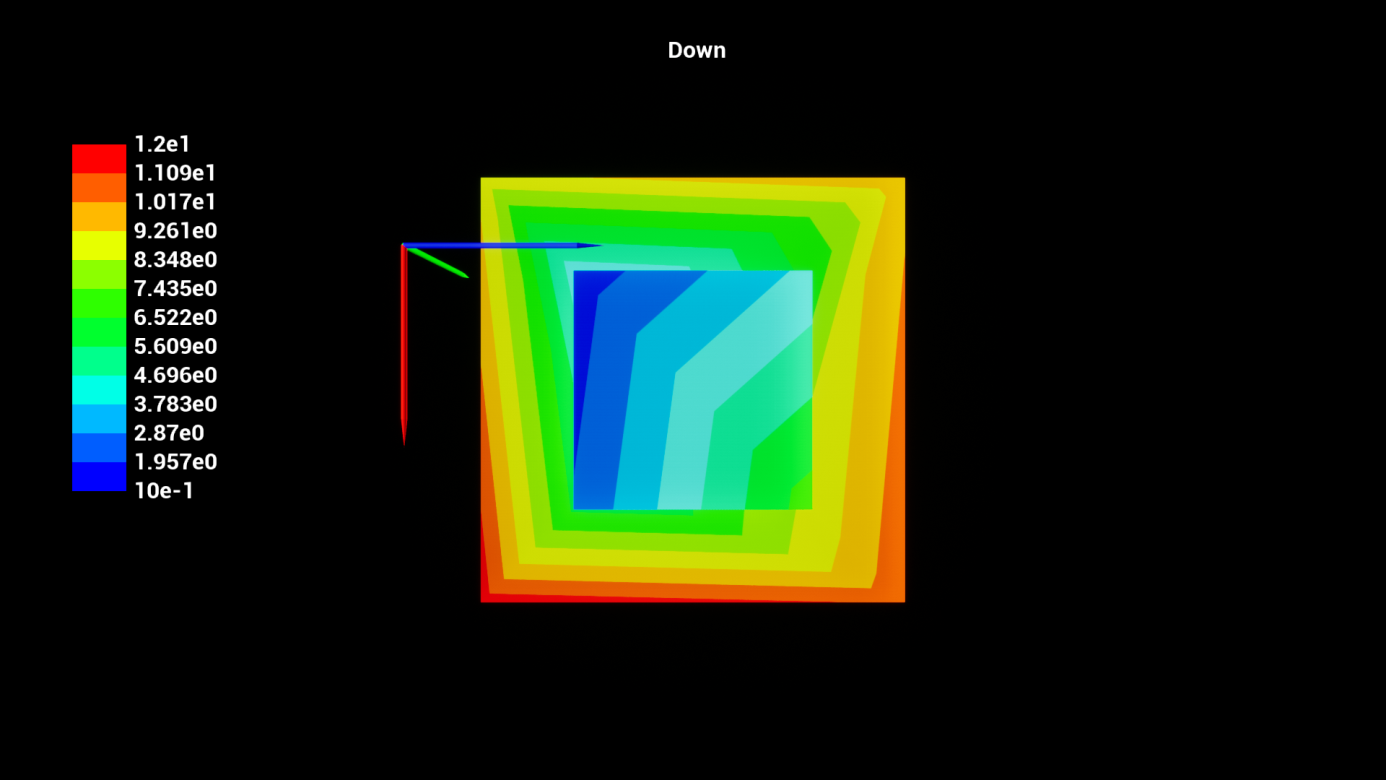
12

NODAL

Przedstawiony powyżej przykład plików jest pokazaniem dwuelementowego modelu o wartościach nałożonych w węzłach oraz typie elementu ośmiowęzłowym. W tym modelu nie zostało uwzględnione przemieszczenie. Wyniki zostały pokazana na **Rys. 1** oraz **Rys. 2**.

****

Rys. 1 Widok przykładowego modelu z tyłu.

****

Rys. 2 Widok przykładowego modelu od dołu.

**Przykładowe zdjęcia z programu.**