POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Multimedia

Animacja 3D

Magdalena Żelezik
Nr albumu s15645

„Co nosimy” – prezentacja informacji z wykorzystaniem animacji 3D i języka JavaScript z WebGL API

Praca inżynierska

Mgr. Piotr Pawłowski

Warszawa, czerwiec 2020

Streszczenie pracy dyplomowej

Praca inżynierska o tytule *„Co nosimy" – prezentacja informacji z wykorzystaniem animacji 3D i języka JavaScript z WebGL API* jest opisem wykonania strony internetowej o charakterze informacyjnym, gdzie tekst został zilustrowany przy wykorzystaniu technik 3D. Tematyka projektu dotyczy zagrożeń środowiskowych i społecznych związanych z przemysłem modowym na świecie. Modele i animacje 3D odwołują się w swoich kształtach i formie do ubrań, tekstyliów oraz pozostałości po nich.

Kod pracy powstał głównie w języku JavaScript z wykorzystaniem biblioteki React. Zostały w nim stworzone sceny i animacje trójwymiarowe w oparciu o standard WebGL, który umożliwia wydajne przetwarzanie grafiki 3D w przeglądarce internetowej. Modele bazowe oraz mocno skomplikowane animacje przygotowane zostały w programie Maya.

Praca dokładnie opisuje zamysł projektowy, proces wykonawczy oraz możliwości i ograniczenia dotyczące użycia technik trójwymiarowych w przeglądarce www.

Słowa kluczowe

Grafika 3D, JavaScript, WebGL

Spis treści

1 WSTĘP

Prace nad optymalizacją liczenia i wyświetlania scen i obrazów przygotowanych w trójwymiarze prowadzone są od lat. Nie chodzi tylko o film czy efekty 3D, ale również zastosowanie grafiki 3D w sieci internetowej. Ich wykorzystanie początkowo głównie interesowało twórców gier, ale również osoby zainteresowane budowaniem wirtualnej rzeczywistości w sieci. Klasycznym już przykładem jest system wirtualnych światów *Second Life,* okrzyknięty fenomenem 2007 roku[1]. Jego popularność wiązała się w panującym wówczas trendem na *życie w sieci* –

> (…) ludzie grupowo wynajmowali wyspy *Second Life*, próbując skolonizować cyberprzestrzeń, która nigdy się nie zmaterializowała[2].

3D w sieci wówczas było namiastką wirtualnej rzeczywistości. Dzisiaj, kiedy mamy Oculus Rift czy Microsoft Hololens, wirtualna rzeczywistość nabrała innego, bardziej *realnego* znaczenia. Grafika 3D jest teraz narzędziem, a nie samą przestrzenią działania. W związku z powyższym, przykłady jej wykorzystania są bardziej abstrakcyjne i kreatywne, niż rekreacja istniejącej przestrzeni i postaci.

W projektowaniu interfejsów użytkownika sieci internetowej poprzez „użycie 3D" lub „efekty 3D" rozumie się dzisiaj wykorzystanie języka WebGL, standardu tworzenia grafiki trójwymiarowej przy wsparciu akceleracji sprzętowej. Standard ten zostanie szerzej opisany w dalszej części tej pracy. W tym momencie warto jednak wskazać, że wykorzystanie WebGL nie oznacza jedynie możliwości kreowania scen 3D z modelami, ale również rozmaite efekty wizualne wykorzystywane w nietypowych przejściach stron czy po najechaniu kursorem na obiekt. W tym standardzie możliwości wizualne do uzyskania w sieci są niemalże nieograniczone, bo o ile twórcy stron www dalej muszą się liczyć z ilością danych do pobrania przez klienta albo możliwościami przeglądarek, tak wsparcie sprzętowe wykonywanych obliczeń daje niesamowitą moc i przestrzeń kreacji.

2 CEL I ZAŁOŻENIA PROJEKTU

Celem tej pracy jest przedstawienie możliwości, jakie dostarcza WebGL w projektowaniu i tworzeniu witryn internetowych. Przedmiotem pracy nie jest jedynie prezentacja potencjalnych pomysłów wykorzystania tego standardu, ale przede wszystkim zarysowanie schematów pracy nad tworzeniem tego typu efektów począwszy od modelowania 3D w dedykowanym programie aż po implementację w kodzie JavaScript.

Założenia projektu są następujące:

1. Projekt jest stroną www stworzoną do korzystania przez dowolnego użytkownika sieci

[1] Parisi, Tony, *Aplikacje 3D. Przewodnik po HTML5, WebGL i CSS3*, Helion 2015, str. 9
[2] Ibidem.

2. Projekt jest przykładem tzw. „wizualnej opowieści” (ang. visual storytelling), przedstawia konkretną historię lub raczej prowadzi przez zbiór informacji niczym film animowany
3. Przy przejściu do kolejnych zagadnień projekt wymaga interakcji z użytkownikiem. Wymóg interakcji jest jawnie zakomunikowany użytkownikowi lub wynika ze standardowych zachowań przy korzystaniu z witryn sieci web
4. Projekt pokazuje rozmaite przykłady wykorzystania standardu WebGL oraz grafiki 3D w prezentowaniu skomplikowanych ilustracji wspomagających przekazanie informacji

W niniejszym tekście zostanie dokładnie przedstawiona metodyka pracy wykorzystana w celu spełnienia wyżej określonych założeń projektu. Opis realizacji jest podzielony na trzy części, z których pierwsze dwie stanowią wprowadzenie do trzeciej. Najpierw zostanie naszkicowany temat grafiki 3D, software'u 3D oraz narzędzi, których wykorzystanie miało miejsce przy realizacji projektu. W drugiej części przedstawione zostanie środowisko programistyczne do budowania dobrze funkcjonującej aplikacji webowej, opisane standardy pracy oraz charakterystyka narzędzi. Trzecią częścią będzie spięcie obu powyższych i szczegółowe opisanie procesów implementacji grafiki 3D w kodzie strony internetowej.

2.1 WYKORZYSTANY SOFTWARE

2.1.1 Grafika 3D

1. Maya 2018, Autodesk (*Student Licence*) –złożony, kompleksowy program do modelowania, animacji i renderowania grafiki 3D. Opanowanie programu Maya na poziomie podstawowym zajęło prawie trzy semestry przedmiotu specjalizacyjnego na kierunku informatyka. Maya, poza modelowaniem brył trójwymiarowych oraz animowaniem ich kształtów posiada rozbudowany silnik fizyczny, umożliwiający tworzenie wymyślnych symulacji.
2. Blender 2.82a – całkowicie darmowy program o możliwościach zbliżonych do programu Maya, jednak o zupełnie innym interfejsie.
3. FBX Review, Autodesk – niewielki program stworzony przez firmę Autodesk to podglądu i testowania plików wyeksportowanych z programów graficznych z rozszerzeniem .fbx

2.1.2 Animacja i postprodukcja

1. Davinci Resolve 16, Blackmagic Design –darmowy program do montażu, edycji oraz nagrywania wideo. Rozszerzenie Fusion do tego programu dodatkowo umożliwia import brył 3D oraz animacji 3D, po czym daje narzędzia do edycji tych animacji czy ich koloryzacji lub montażu. Rozbudowana biblioteka efektów pozwala wzmacniać efekt pracy w programach 3D

2.1.3 Implementacja

1. Visual Studio Code, Microsoft – darmowy program do edycji kodu. Posiada wbudowaną konsolę wiersza poleceń oraz szereg rozszerzeń, przykładowo do bieżącego sprawdzania poprawności semantycznej pisanego kodu.
2. Chrome, Google – darmowa przeglądarka od Google, która w swoich nowszych wersjach zapewnia bezproblemowe wykorzystanie standardu WebGL. Jest dodatkowo wyposażona w Chrome Developer Tools, czyli zestaw narzędzi deweloperskich, które pomagają w łatwy sposób wykryć błędy w kodzie.
3. Git, Git Foundation – system kontroli wersji kodu.

2.2 WYKORZYSTANE JĘZYKI PROGRAMOWANIA I BIBLIOTEKI

2.2.1 Języki programowania i znacznikowe

1. JavaScript – to „skryptowy (interpretowany lub kompilowany metodą JIT) język programowania, w którym funkcje są *obywatelami pierwszej kategorii* – obiektami, które można przechowywać w zmiennych jako referencje i przekazywać jak każde inne obiekty”[3] – to cytat za wstępem do dokumentacji JavaScript Mozilla Developer Network, jednej z najpełniejszych dokumentacji tego języka. JavaScript nie należy mylić z Javą, jedyny prawdziwy związek tych dwóch języków to to, że obydwa są znakami towarowymi firmy Oracle[4]. JavaScript wykorzystuje się w programowaniu aplikacji webowych, gdyż kompilator tego języka jest standardem każdej przeglądarki. JavaScript, jak według powyższego cytatu, opiera się na wykorzystaniu funkcji jako podstawowej jednostki.
2. CSS3 – system znacznikowy do zapisu styli obowiązujących na stronie www. CSS to z języka angielskiego kaskadowe arkusze styli (*cascade style sheets*). Uszczegóławiając: style to właściwości wyświetlanych elementów, typu pozycja, kolor, margines. Arkusz to inna nazwa pliku ze znacznikami i odpowiadającymi im parametrami i ich właściwościami. Kaskadowy oznacza priorytetyzację wykonania konkretnych zapisów. Jeśli coś jest zapisane wyżej w arkuszu, może zostać nadpisane wyżej. O priorytecie zapisu decyduje również szczegółowość znacznika. CSS3 to najnowszy standard CSS obsługujący również wydajne animacje elementów, również w przestrzeni 3D.
3. HTML5 – język znacznikowy wykorzystywany do zapisu układu elementów na stronie oraz ich wstępnej charakterystyki. W HTML są elementy odpowiedzialne za przechowywanie tekstu, obrazu lub zbiorów innych elementów. Poprawna semantyka HTML jest ważna zwłaszcza dla silników wyszukiwarek indeksujących witryny dla tysięcy zapytań dziennie. HTML5 to najnowszy standard HTML obsługujący m.in. dodatkowe zapisy ułatwiające korzystanie z sieci

[3] MDN web docs, https://developer.mozilla.org/pl/docs/Web/JavaScript
[4] Ibidem.

dla osób z niepełnosprawnościami oraz wydajniejszą obsługę materiałów multimedialnych, takich jak wideo, ale również obraz generowany bezpośrednio w kodzie strony internetowej.

4. GLSL – język skryptowy do zapisu operacji wykonywanych przy akceleracji sprzętowej karty graficznej maszyny, na której wyświetlana jest strona. GLSL (*graphic library scripting language*) jest wykorzystywany przy tworzeniu shader'ów (szerzej opisanych w dalszej części pracy).

2.2.2 Biblioteki

1. Three.JS – biblioteka oparta o standard WebGL. Za pomocą Three można pominąć trud uczenia się WebGL, który ma dosyć skomplikowaną składnię oraz niską możliwość kontroli błędów w środowisku przeglądarki i posługiwać się tym językiem przy pomocy funkcji JavaScript. Three pozwala w łatwy sposób budować sceny 3D oraz animacje. Twórcy Three opracowali bardzo szczegółową dokumentację biblioteki opatrzoną przykładami, z których kilka jest zreprodukowanych w tej pracy.
2. PIXI.js – biblioteka również oparta o standard WebGL, jednak z mniejszym ukierunkowaniem na budowanie scen i animacji 3D jak Three, a większym na proste tworzenie efektów wizualnych opartych o shadery.
3. GSAP, GreenSock – biblioteka do pracy z animacją na stronie. GSAP zawiera szereg narzędzi regulujących czas i kolejność animacji, z poszanowaniem standardów różnych przeglądarek. Dodatkowo zapewnia jak najsprawniejsze działanie animacji przez wykorzystanie do ich obsługi transformacji CSS, które są najwydajniejsze do tego typu operacji.
4. ScrollMagic.js – biblioteka pozwalająca skutecznie i bezbłędnie monitorować przesuwanie się po stronie przez użytkownika i na bazie informacji o jego obecnej pozycji wyzwalać animacje lub pokazywać elementy.
5. React.js – potężna biblioteka autorstwa firmy Facebook umożliwiająca budowanie szybkich i wydajnych aplikacji typu SPA (ang *single-page application*, aplikacja jednostronicowa) w oparciu o moduły. React opisany będzie szeroko w drugiej części dotyczącej realizacji projektu.

3 GRAFIKA TRÓJWYMIAROWA – KRÓTKA CHARAKTERYSTYKA

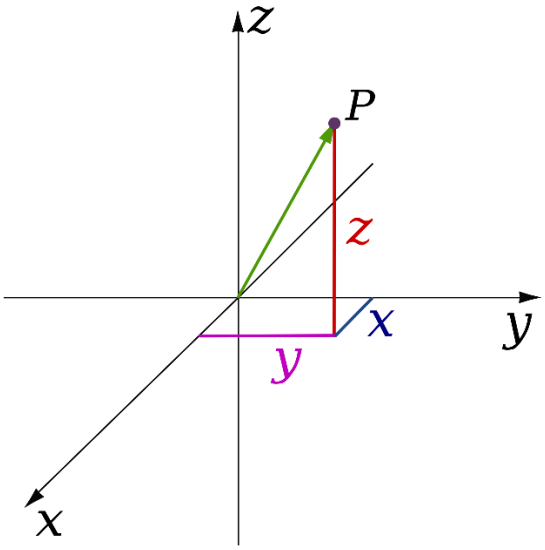
3.1.1 Wprowadzenie

Dokładne zagłębianie się w charakterystykę matematyczną grafiki trójwymiarowej nie jest przedmiotem tej pracy. Należy jednak dla spójności przekazu wyjaśnić kilka podstawowych pojęć oraz nomenklaturę, która jest używana w tym tekście. Cytując za przytoczoną wcześniej książką Tony'ego Parisi[5]:

> Komputerowa grafika trójwymiarowa (w odróżnieniu od dwuwymiarowej) to grafika wykorzystująca trzy wymiary do reprezentacji danych geometrycznych (często kartezjańskich), które są przechowywane w komputerze w celu wykonywania obliczeń i renderowania obrazów dwuwymiarowych. Obrazy takie można przechowywać w celu wyświetlenia w odpowiednim momencie albo wyświetlać na bieżąco.

Pozwolę sobie, podobnie jak autor cytowanego tekstu, rozbić tę definicję i objaśnić jej poszczególne elementy.

Co to znaczy, że grafika wykorzystuje trzy wymiary? W dużym skrócie można to wyjaśnić w ten sposób, że każdy z punktów na płaszczyźnie w trójwymiarze jest zapisywany przy użyciu trzech współrzędnych. Powyższe definicja przywołuje układ kartezjański i najczęściej spotkamy się z takowym podczas pracy z grafiką 3D:



Rysunek 1. Kartezjański układ trzech współrzędnych[6]

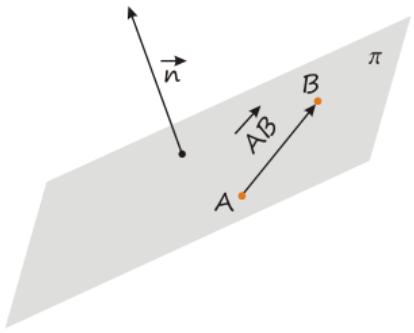
Płaszczyzna, do której należy punkt, może być albo częścią sceny albo jedną ze ścian bryły w scenie trójwymiarowej. W matematyce modeli trójwymiarowych nie jest zapisywany jednak każdy punkt na każdej płaszczyźnie. Przechowuje się współrzędne tych punktów, które są końcami odcinków między którymi rozpostarta jest płaszczyzna.

[5] Parisi, Tony, *Aplikacje 3D. Przewodnik po HTML5, WebGL i CSS3*, Helion 2015, str. 22
[6] Andeggs - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=7504188

Inną ważną własnością płaszczyzn, o której będzie mowa w tej pracy, jest wektor normalny. Wektorem normalnym jest wektor prostopadły do płaszczyzny. Łatwa do zrozumienia definicja wektora normalnego płaszczyzny znajduje się w e-podręczniku do matematyki AGH w Krakowie[7]:

> Wektor $\vec{n}$ jest prostopadły do płaszczyzny π, jeżeli dla dowolnych dwóch jej punktów A i B wektory $\overrightarrow{AB}$ oraz $\vec{n}$ są prostopadłe.
>
> Każdy niezerowy wektor prostopadły do płaszczyzny nazywamy wektorem normalnym tej płaszczyzny.

Ilustruje to poniższa rycina:



Rysunek 2 Wektor normalny[8]

Wektory normalne płaszczyzn wykorzystuje się w np. w obliczeniach dotyczących ilości światła padającego na płaszczyznę i odbitego, przesłaniania się wzajemnego brył oraz wielu innych.

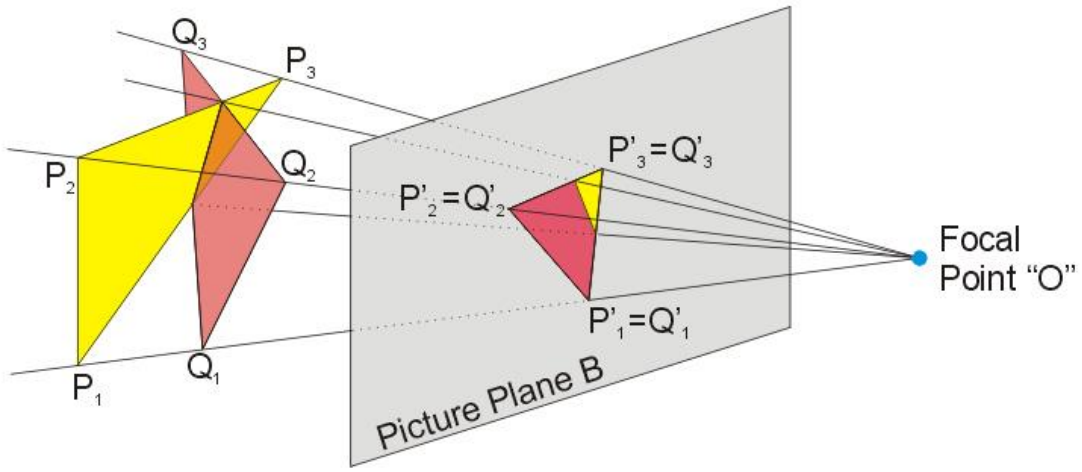
Posiadając takie podstawowe informacje jasne jest, że to właśnie między innymi te własności modeli 3D są wykorzystywane do obliczeń ich wzajemnego położenia, kolorów czy jasności. W tym miejscu istotne jest wyjaśnienie pojęcia renderowania.

Renderowanie jest kalką z języka angielskiego i nie ma swojego polskiego odpowiednika. W swoim ogólnym znaczeniu renderowanie to interpretacja treści elektronicznych w celu prezentacji ich w sposób odpowiedni dla swojego środowiska przeznaczenia. Przykładowo, podczas pisania tej pracy w programie Microsoft Word widok „kartki papieru" na której piszę jest renderem na ekranie jakiegoś zapisu cyfrowego o białym prostokącie.

W temacie grafiki trójwymiarowej, kontynuując objaśnianie definicji, renderowanie oznacza interpretację obrazu trójwymiarowego na finalnym dwuwymiarowym obszarze wyświetlania. Dwuwymiarowy zapis obrazu sceny 3D to zapis jej projekcji na płaszczyznę w stosunku do centralnego punktu wzrokowego. W dużym skrócie celem jest zarejestrowanie tylko tych fragmentów, które „widać" z danego kąta obserwacji sceny. Szczegółowe objaśnienie procesu renderowania nie jest przedmiotem tej pracy, ale rycina poniżej pozwala zrozumieć podstawowy problem do rozwiązania w algorytmach

[7] Góra, Michał, Matematyka, E-podręcznik AGH, https://epodreczniki.open.agh.edu.pl/tiki-index.php?page=P%C5%82aszczyzny+w+tr%C3%B3jwymiarowej+przestrzeni+rzeczywistej
[8] Ibidem.

obsługujących projekcję, tj. zapis w przestrzeni dwuwymiarowej tylko tych punktów należących do danych brył, które mogą zostać wyświetlone.



Rysunek 3 Projekcja kompozycji 3D na płaszczyznę B[9]

Kolejnym problemem renderowania, który na tej rycinie jest już rozwiązany, ale jest mocno skomplikowany, jest problem światła i cienia, wzajemnego oświetlenia brył oraz ich ustawienia względem oka. Renderer musi oszacować, który element znajduje się bliżej oka, a który dalej, zapis samych punktów może nie wystarczyć.

Wracając do definicji grafiki trójwymiarowej należy jeszcze przyjrzeć się ostatniemu zdaniu. Przechowanie obrazu do wyświetlenia w odpowiednim momencie jest dosyć jasne, przykładowo może chodzić o klatki animacji w filmie 3D. Wyświetlanie na bieżąco jest tematem bardziej skomplikowanym. Powyżej, bardzo pobieżnie, zostały opisane matematyczne zależności dotyczące bryły 3D. W zasadzie pominęłam kwestie oświetlenia i teksturowania, które, wykonane poprawnie, nadają bryłom realizmu. Jak zapewnić, by ta mnogość operacji była w stanie wykonać się tak szybko, by móc wyświetlać animację 3D na bieżąco? Ten temat będzie się pojawiał przy objaśnianiu kolejnych przykładów produkcji scen 3D w programie, a także przy użyciu WebGL.

3.1.2 Nazwy i pojęcia

W tej pracy pojawi się wiele nazw, które należy objaśnić przez rozpoczęciem korzystania z nich. Wiele z nich ma swoje odpowiedniki w języku polskim (podane poniżej kursywą), ale w użyciu profesjonalnym używa się dla wygody nazw angielskich i mogę z nich korzystać wymiennie w dalszych partiach tego tekstu. Mogą też pojawiać się na rycinach, w kodzie lub w zdjęciach interfejsu programów graficznych.

[9] Źródło i licencja: CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=337435

1. Vertex (*wierzchołek*) – punkt w przestrzeni 3D. Pomiędzy wierzchołkami rozpięte są odcinki, które najczęściej wyznaczają brzegi płaszczyzny będącej częścią składową bryły. Wierzchołki mogą mieć też obiekty dwuwymiarowe w przestrzeni 3D np. krzywe.
2. Edge (*brzeg*) – odcinek wyznaczony przez dwa wierzchołki. Pomiędzy odcinkami może rozpięta być płaszczyzna
3. Face (*ściana*) – płaszczyzna, będąca częścią składową bryły 3D. Ściany zawsze są wielokątami, najczęściej czworokątami lub trójkątami
4. Mesh (*siatka*) – dwie lub więcej ściany połączone ze sobą krawędziami. Siatka jest opisem bryły (modelu) w przestrzeni trójwymiarowej
5. Geometria – zbiór danych dotyczących wierzchołków, wektorów normalnych, pozycji bryły.
6. Materiał – wygląd ściany bryły, tak naprawdę zapis właściwości, jakie musi wziąć pod uwagę renderer, interpretując daną bryłę, przykładowo: kolor, ilość odbijanego światła, refleksy etc.
7. Tekstura –obraz (ze zdjęcia, rysunek, wygenerowany cyfrowo) nakładany na materiał w celu uzyskania konkretnych efektów wizualnych
8. Scena – zbiór elementów 3D oddziałujących na siebie wzajemnie
9. Kamera – punkt widzenia na scenę 3D, wykorzystywany przy renderowaniu
10. Renderer – program lub funkcja interpretująca scenę 3D na obraz 2D

4 CHARAKTERYSTYKA PROJEKTU GRAFICZNEGO

4.1 DZIEDZINA PROBLEMOWA

Warto stanąć w korytarzu dowolnego centrum handlowego, najlepiej w drugiej połowie dnia w weekend i zwrócić uwagę na krążące wokół tłumy. Następnie wejść do dowolnego sklepu, najlepiej odzieżowego i spojrzeć na uginające się od towarów półki i wieszaki. Później, zanim dostanie się bólu głowy od hałasu i nadmiaru bodźców wzrokowych, zadać sobie kilka pytań, wszystkie zaczynające się od: ile? Ile trzeba wyprodukować obiektów, aby przy takim popycie półki nadal wydawały się pełne? Ile różnych sklepów może sprzedawać ten sam produkt, nie tracąc na tym zysku? Ile ubrań, butów, rajstop i zegarków potrzebować może jedna osoba? Ile towarów na koniec miesiąca nie znajdzie nowych właścicieli?

Jeszcze przed rozpoczęciem badań do pracy wiedziałam, że jest coś głęboko nieetycznego w tiszercie za 15 złotych w dwupaku z napisem CONSCIOUS (ang. świadomy, w domyśle chodzi o świadomość kosztu ekologicznego produkcji). Domyślałam się też, że jest coś patologicznego w tym, że ludzie wydają pieniądze, których nie mają na rzeczy, których nie potrzebują i które nawet im się nie podobają[10].

Research do projektu wykonał się prawie sam. Zaczęłam od obejrzenia filmu True Cost z 2015 roku, a na bazie usłyszanych tam informacji wystarczyło wpisać odpowiednie hasła w wyszukiwarkę, by utonąć w morzu rekordów. Rynek tzw. "szybkiej mody" (ang. fast fashion) jest jednym z najgorszych w kwestii zanieczyszczania środowiska[11].

4.2 WIZUALNA OPOWIEŚĆ

Skala problemu jest tak duża, a waga pozyskanych informacji tak ciężka, że pomysł zebrania tych liczb i haseł w całość, nadania im ekspresyjnej formy i udostępnienia jak największej liczbie odbiorców wydaje się czymś oczywistym. Wybrałam w tym celu formę „wizualnej opowieści" (ang. *visual storytelling*), gdyż z doświadczenia wiem, iż jest czymś bardziej pochłaniającym niż suche fakty przedstawione przy pomocy tekstu. Wizualna opowieść może być częścią filmu czy animacji, ale może też być częścią tekstu. W tym ostatnim podejściu nie chodzi o zdania opatrzone ilustracjami, ale raczej o obrazy z dodatkiem niewielkiej ilości słów dla podkreślenia swojej wagi.

W projekcie strony pojawiają się trzy części główne i wiele mniejszych. Główne partie to: wstęp, gdzie pojawia się kilka liczb ze statystyk, animowanych dla podkreślenia dramatyzmu. Wstęp ma przykuć uwagę odbiorcy do liczb pojawiających się w całej treści strony, nawet kiedy towarzyszące im obrazy będą znacznie ciekawsze. Kolejną partią są sekcje problemowe: ilość ubrań na osobę, masa

[10] Parafraza za: 1928 June 4, The Detroit Free Press, Paragraphs by Robert Quillen, Quote Page 6, Column 4, Detroit, Michigan.

[11] https://www.theguardian.com/commentisfree/2019/jun/18/fast-fashion-environmental-audit-committee-polluting-industry

produkcji branży modowej, wyzysk pracy na początku łańcucha dostaw, niska jakość tekstyliów, toksyny wykorzystywane przy pozyskiwaniu materiałów i ich wpływ na środowisko. Ostatnią partią jest sekcja źródeł i przydatnych linków, skąd odwiedzający może zaczerpnąć więcej wiedzy i uzyskać inspirację do bardziej świadomej konsumpcji. Ta część jest bardzo istotna, gdyż zdaję sobie sprawę, że przedstawienie poszczególnych zagadnień w sposób ciekawy wizualnie sprawia, że można przekazać tylko procent informacji, które są niezmiernie istotne w opisie tego problemu. Dodatkowo jest to swoisty podział na wstęp, rozwinięcie i zakończenie, jak w przykładnej opowieści przystało.

5 REALIZACJA PROJEKTU

5.1 GRAFIKA 3D

5.1.1 Modele ubrań

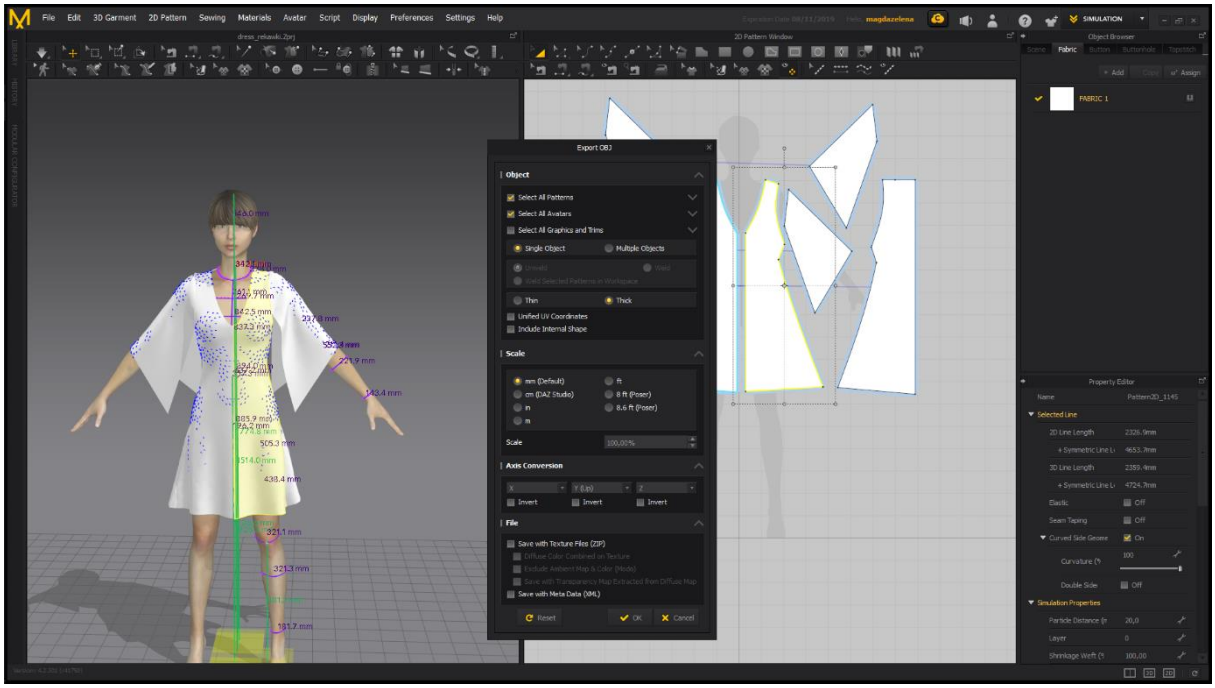
W projekcie pojawia się jedenaście różnych modeli sukienek. Sukienki są symbolami wszystkich ubrań ogólnie, wybrałam je jako przykład, gdyż to do kobiet najczęściej kierowane są reklamy, którymi bombardują nas sklepy.

Modele sukienek zostały wykonane przy pomocy programu Marvelous Designer 8. Jest to program dedykowany do modelowania wzorów odzienia. Praca z nim jest nieco odmienna od typowej pracy z modelowaniem 3D. Aby wykonać poprawną formę okrycia należy, niczym prawdziwa krawcowa, przygotować odpowiedni wykrój ubrania, dodać szwy (określić wcześniej ich rodzaj, elastyczność). Trzeba wybrać również rodzaj i strukturę materiału, czy ma być elastyczny czy sztywny, ciężki czy lekki. O ile interfejs programu jest bardzo intuicyjny, najdłużej zajęła mi nauka przygotowania poprawnych wykrojów ubrań.

Kolejność wykonywanych zadań w celu uzyskania poprawnego modelu w Marvelous Designer 8 jest następująca:

1. Przygotowanie wykroju ubrania
2. Umieszczenie elementów wykroju dookoła modelu awatara – przykładowo front bluzki z przodu klatki piersiowej
3. Na wykroju zaznaczenie brzegów, które mają zostać zszyte razem, a następnie dodanie w tym miejscu szwów
4. Po ukończonych przygotowaniach kliknięcie „play” nad obrazem awatara, które włącza „fizykę” – materiał opada w zgodzie z grawitacją, ale szwy ściągają połączone brzegi do siebie. Automatycznie buduje się uszyte ubranie
5. Fałdy materiału można przesuwać na modelu dokonując „poprawek stylistycznych”
6. Gotowe ubranie eksportuje się jako trójwymiarowy model do użycia w innych programach graficznych

Poniżej znajduje się zapis obrazu z przygotowania modelu jednej z sukienek:



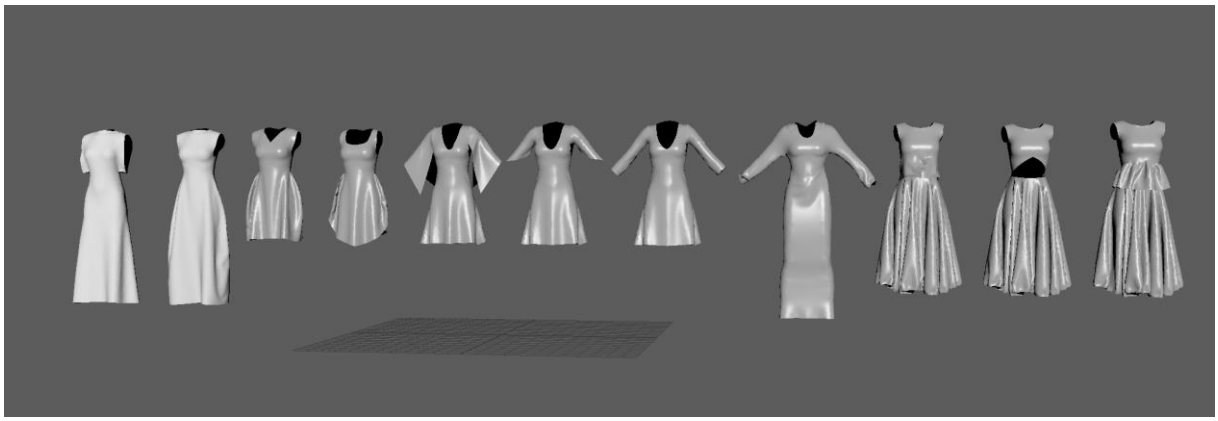
Rysunek 4 Widok z programu Marvelous Designer

Po prawej stronie widzimy ekran wykroju. U jego góry widoczne są niewielkie ikonki z narzędziami do tworzenia wielokątów, modelowania krzywych czy wycinania dziur. Nieco niżej znajdują się ikony maszyn do szycia, które odpowiedzialne są za dodawanie szwów we wcześniej zaznaczonych miejscach wykroju.

Lewa strona przedstawia obraz awatara z już nałożonym i zamodelowanym wykrojem sukienki. Niebieskie kropki na powierzchni materiału to miejsca styku ubrania z ciałem. Im większa gęstość kropek, tym ubranie jest ciaśniejsze na awatarze – pozwala to odpowiednie dobranie rozmiarów na wykroju. Na gotowej sukience oznaczone są też wszystkie miary ubrania oraz miary względem wysokości postaci – mamy dostępne awatary różnej płci, postury czy wieku.

Na środku widać okno ustawień do eksportu modelu do formatu .obj (Object 3D). Sukienkę eksportowałam jako Single Object, czyli pojedynczy obiekt. Można też wybrać Multiple Objects, kiedy zamodelowana jest np. bluzka i spódnica. Wyeksportują się wtedy jako dwie osobne bryły.

Postępując według wyżej opisanego schematu zadań stworzyłam jedenaście różnych modeli sukienek, które zaimportowałam w formacie .obj do programu Maya:



Rysunek 5 Modele sukienek zaimportowane do Maya 2018

5.1.2 Właściwości fizyczne tkanin

O ile zachowanie właściwości fizycznych tkanin jest podstawą działania programu Marvelous Designer i jest integralną częścią działania programu, w Maya oraz w innych środowiskach do pracy z 3D należy posiadać pewną wiedzę, aby osiągnąć zbliżony efekt.

Warto jednak zauważyć w tym miejscu, że choć tkanina jest de facto, materiałem, z którego wykonany jest jakiś obiekt, w pracy z grafiką 3D nie należy traktować tych dwóch pojęć jako tożsamych. Tak, jak to było wskazane powyżej w słowniku pojęć, materiał bryły 3D to zestaw cech face'ów niezbędnych przy renderowaniu. Materiał w tym kontekście przekaże informacje o kolorze, odbiciu światła etc. Materiał nie ma możliwości dodania cech fizycznych obiektowi, na który jest nałożony. O ile, przykładowo, będziemy chcieli imitować stal, możemy w materiale nadać ścianom srebrny kolor i wysoką połyskliwość, ale nie nadamy twardości czy trwałości czy wagi. Takie właściwości zaaplikujemy równolegle na sam mesh.

W Maya jest wygodne narzędzie do nadawania własności tkanin siatce obiektu. Nazywa się nCloth i, według definicji tłumaczonej z języka angielskiego z dokumentacji programu, jest to:

> (..) szybkie i stabilne rozwiązanie do tkanin, które wykorzystuje system połączonych cząsteczek do symulowania szerokiej gamy dynamicznych powierzchni wielokątów. Na przykład, nCloth jest wystarczająco elastyczny do symulacji następujących substancji: ubrań tekstylnych, nadmuchiwanych balonów, powierzchni tłukących się oraz obiektów deformowanych[12].

Rozszerzając nieco tę definicję, nCloth pozwala na dodanie obiektu do przestrzeni, w której oddziaływają na niego prawa fizyki, takie jak grawitacja oraz dodatkowo rozszerza jego własne właściwości o takie parametry jak waga, ciągliwość, elastyczność, trwałość etc. Dodatkowo symulator nałożony na siatkę sugeruje się jej pierwotnym kształtem, ale wpływa na poszczególne elementy siatki w celu uzyskania konkretnych efektów.

[12] Dokumentacja Maya, https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2020/ENU/Maya-CharEffEnvBuild/files/GUID-ED791F1C-8412-4785-829F-9925F2604E8A-htm.html, tłumaczenie własne

W projekcie użyłam nCloth do uzyskania dwóch efektów. Pierwszy z nich to eksplozja sukienki. Przygotowane wcześniej modele ubrań chciałam transformować jedno w drugie. Nie jest to jednak możliwe przy pomocy klasycznego przekształcania brył za pomocą Maya, gdyż siatki posiadają różną liczbę ścian. Modele są na tyle różne, że ręczne dopasowanie liczb face'ów do siebie nie wchodziło w grę, dodatkowo uzyskanie satysfakcjonujących rezultatów tą metodą mogłoby okazać się niemożliwe. Innym rozwiązaniem jest zamiana jednego obiektu na drugi. Aby uzyskać efekt przejścia między modelami, pierwszy obiekt będzie niszczony do chmury małych fragmentów, z których następnie formować będzie się kolejny.
W pierwszym podejściu do realizacji tego pomysłu zaczęłam od „rozpadu" obiektu sukienki na cząsteczki - particles, przy użyciu generatora particles w Maya. Umieściłam słowo rozpad w cudzysłowie, gdyż tak naprawdę cząsteczki nie są częścią siatki, ale zupełnie oddzielnymi bytami. Taki pseudo-rozpad można uzyskać wybierając siatkę jako pierwotny kształt chmury cząstek, a sam materiał siatki ustawiając na niewidoczny. Efekty są ciekawe, ale w trakcie pracy sprawdziłam, że particles nie są możliwe do wyeksportowania jako animacja i moja kontrola nad nimi byłaby ograniczona. Generowanie ich przez JavaScript tą metodą do uzyskania satysfakcjonujących mnie efektów nie byłoby dostatecznie wydajne.
Drugie podejście zakładało faktyczny rozpad obiektu na małe cząstki - najlepiej po linii brzegowej face'ów. To rozwiązanie po pierwsze zapewniłoby mi możliwość eksportu animacji jako obiektu, a po drugie zachowałoby efekt operowania na bezpośrednio na tkaninie i nadało wrażenie realności. Właśnie ten proces opiszę w szczegółach w następnym podrozdziale.

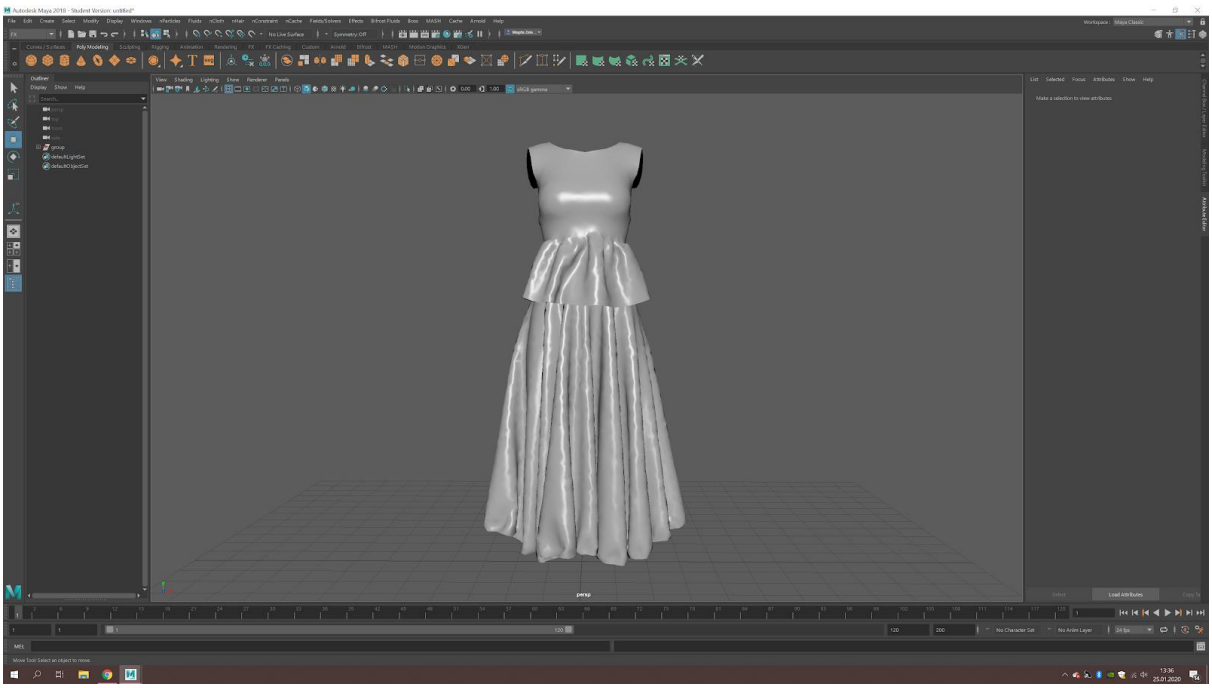
Drugi efekt do którego wykorzystałam nCloth to symulacja poruszania się ubrania bez postaci w środku, na niewidocznym wieszaku. Sam ten proces jest opisany poniżej i jest o wiele mniej skomplikowany niż eksplozja, ale trudności dostarczył eksport brył z zachowaniem animacji materiału, jednak bez silnika fizycznego.

5.1.3 Eksplodowanie sukienki

Proces pracy był następujący:

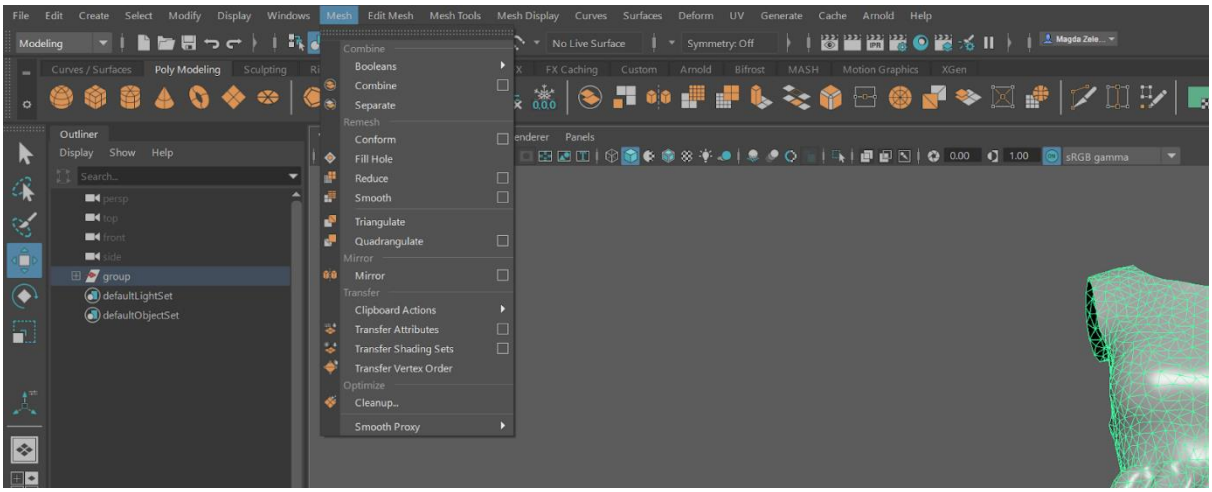
1. Obiekt otrzymuje cechy tkaniny nCloth, usuwam z otoczenia grawitację, by fizyka dotyczyła tylko struktury ścian sukienki,
2. Nadaję wszystkim brzegom face'ów charakter „rozrywanej powierzchni" - w razie kolizji z innym obiektem mam kontrolę nad tym, gdzie przerwie się materiał,
3. W środku obiektu umieszczam inny - niewielki prosty model geometryczny, jego tekstura będzie przeźroczysta, wobec czego sam model niewidoczny,
4. Bryle nadaję cechy obiektu pasywnego, a więc będzie miała cechu fizyczne i będzie mogła wchodzić w kolizję z innymi obiektami,
5. Animuję rozmiar bryły w czasie od niewielkiego do o wiele wykraczającego poza obszar modelu sukienki.

Dzięki nadaniu odpowiednich cech tkaniny modelowi sukienki, bryła powinna rozerwać go od środka. Ponieważ sukienki miały różne formy, do kolizji z nimi użyłam również różnego rodzaju brył. Poniżej przykład krok po kroku rozerwania materiału za pomocą bryły walca:

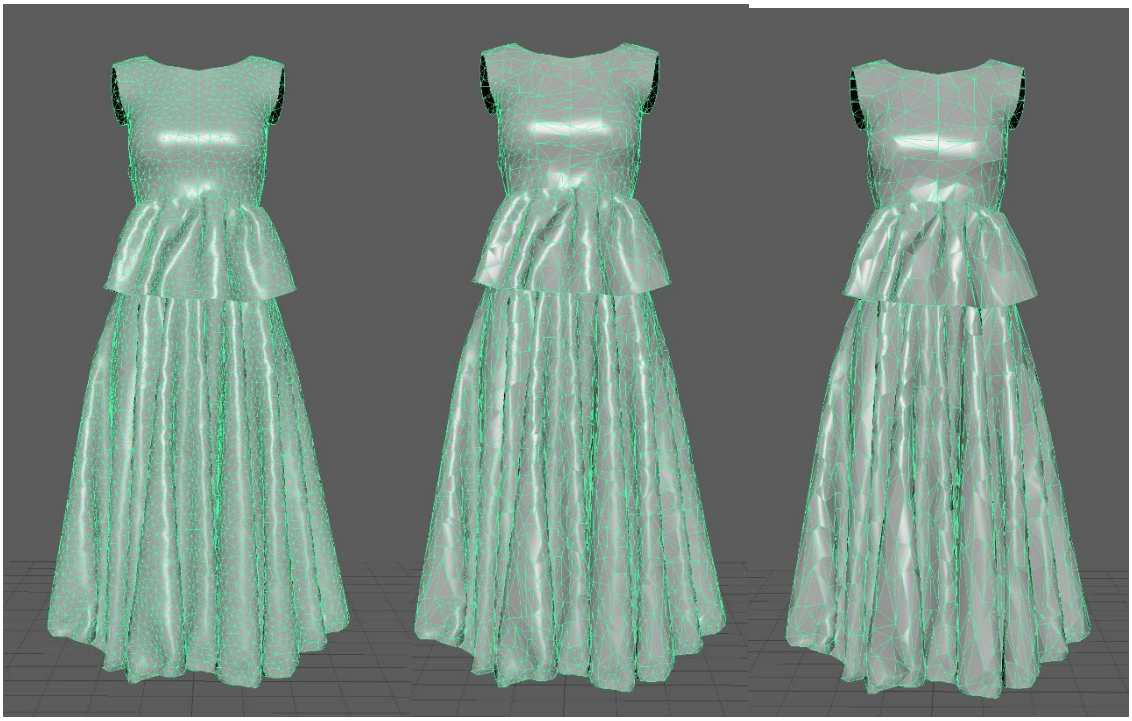


Rysunek 6 Model sukienki w programie Maya

Wykonany przy pomocy Marvelous Designer model sukienki umieszczam w nowej scenie w Maya. Model jest stosunkowo realistyczny, co oznacza, że składa się z ogromnej liczby face'ów. Obliczenie trajektorii lotu i deformacji dla każdej z nich będzie niezwykle kosztowne, więc należy pójść na kompromis i zrezygnować z części szczegółów na rzecz interesującego efektu. Korzystając z narzędzia Mesh->Reduce redukuję liczbę ścian modelu aż do uzyskania takiej formy, która jeszcze eksponuje część pierwotnych szczegółów, ale będzie optymalna do późniejszej animacji.

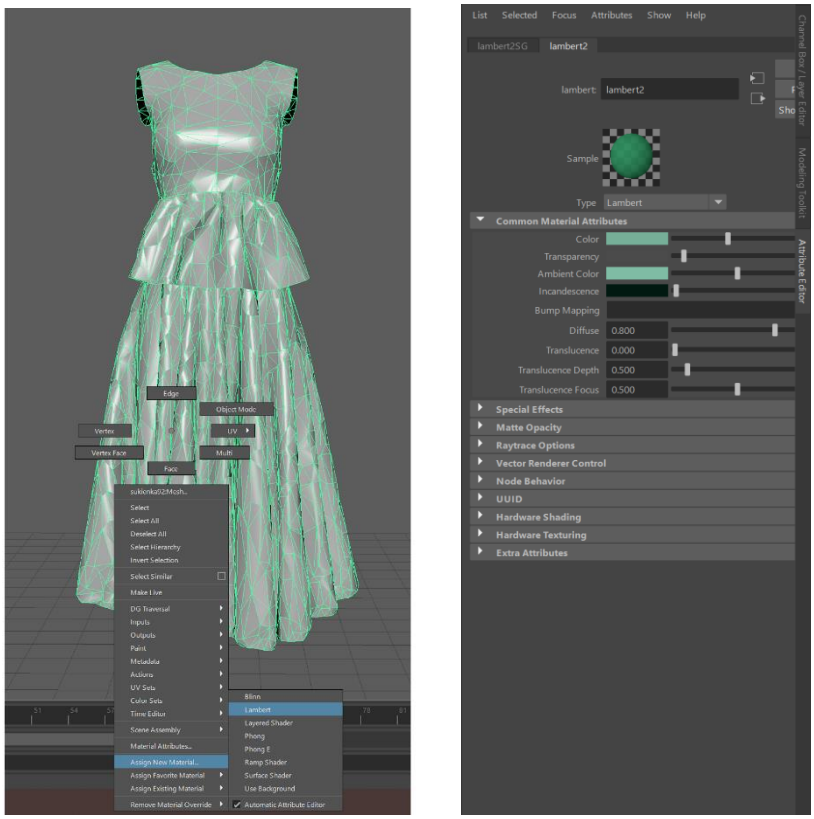


Rysunek 7 Opcja Mesh->Reduce znajduje się w menu Modeling



Rysunek 8 Wygląd siatki po kolejno wykonanych redukcjach

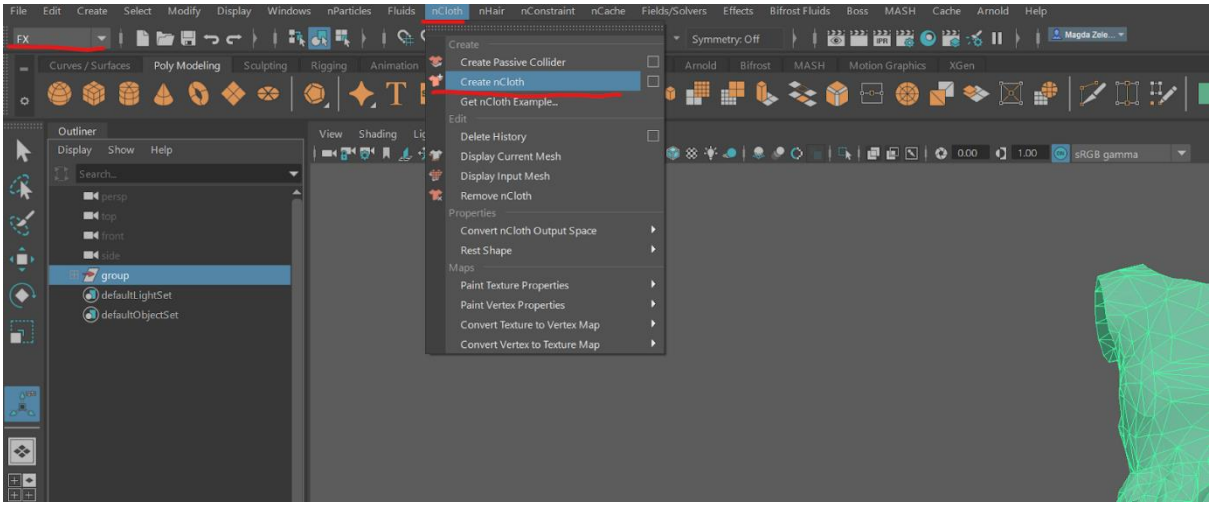
Niestety widać znaczną różnicę w jakości końcowej bryły. Dodanie odpowiedniej tekstury zredukuje nieco nieprzyjemny efekt kanciastości. Wybieram materiał Lambert, dzięki któremu łatwo uzyskam matową powierzchnię, a zależy mi na złagodzeniu kantów i odbić. Tekstura, którą dodaję, jest jasna, nieco przezroczysta, odbijająca światło. Załamania bryły, zwłaszcza na fałdach sukni stają się mniej widoczne, trochę rozmyte, ale zważywszy na fakt, że model będzie się poruszać, taki efekt jest jak najbardziej akceptowalny.



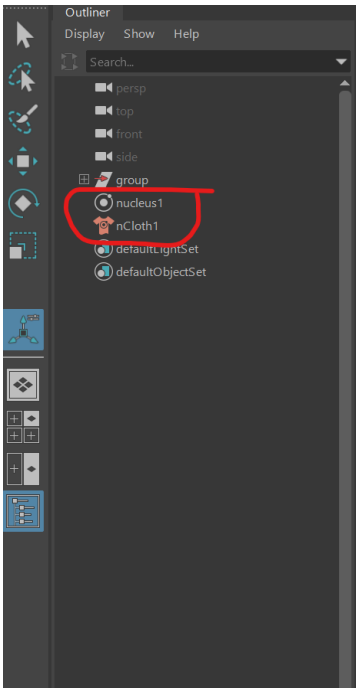
Rysunek 9 Proces dodania nowego materiału do siatki oraz przykład ustawień materiału

Rysunek 10 Model po redukcji liczby ścian i nadaniu materiału

Następnie model potrzeba zmienić na materiał nCloth. W tym celu na pasku narzędzi przełączę się z menu modelowania na menu FX i wybiorę odpowiednią funkcję z menu nCloth:

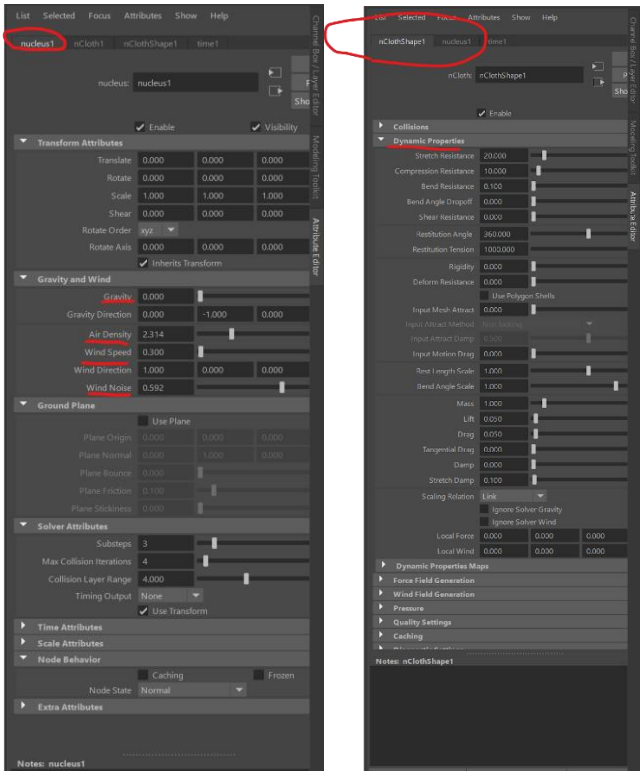


Rysunek 11 Utworzenie symulatora nCloth na modelu

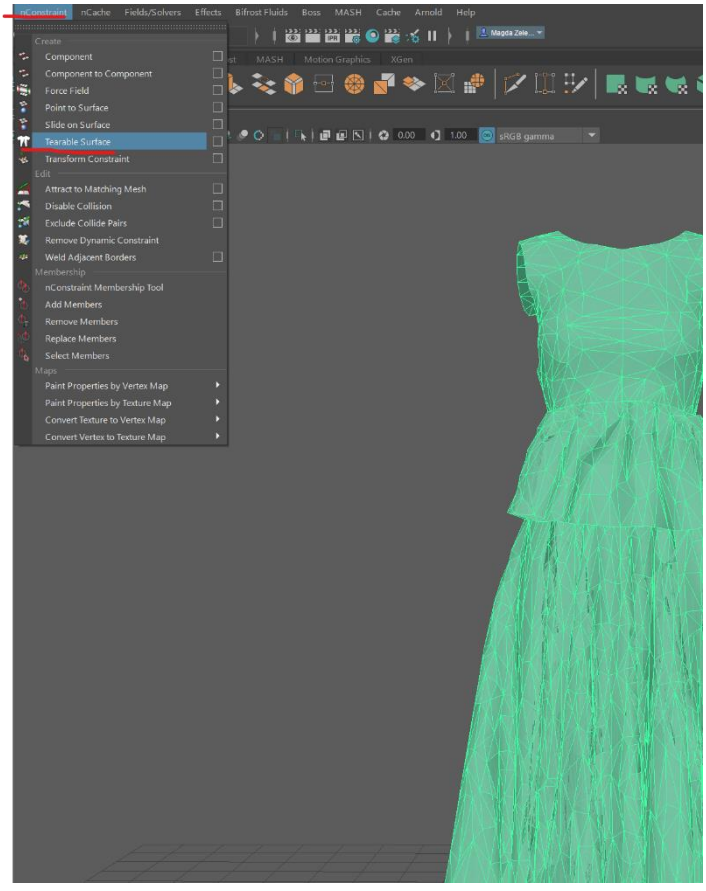


Rysunek 12 Obiekty po utworzeniu symulatora

Dodane zostały dwa nowe obiekty, takie jak nucleus i sam nCloth. Nucleus to węzeł pozwalający na wykonywanie matematycznych obliczeń związanych z fizyką innych węzłów, takich jak m.in. nCloth albo nParticles. Dzięki niemu można kontrolować takie właściwości jak np. wiatr czy grawitacja oddziaływująca na obiekt. Aby model sukienki nie opadał w dół, należy zmienić wartość siły grawitacji z domyślnej 9,8 na 0. Dodatkowo warto będzie wskazać na charakter materiału obiektu, więc można nadać niewielką wartość sile wiatru, aby fałdy sukienki delikatnie falowały. Cechy samego materiału można edytować w zakładce nClothShape wybierając właściwości dynamiczne (*Dynamic Properties)*. Zwiększenie wartości odporności na rozciąganie i zginanie (odpowiednio *Streach Resistance* i *Bend Resistance*) sprawi, że sukienka rozerwie się, a nie rozciągnie na walcu jak guma. W celu osiągnięcia ciekawych efektów animacji można też nieznacznie zwiększyć wartość trwałości (*Rigidity)* o dosłownie 0.004 jednostki - wówczas po rozpadzie fragmenty będą walczyć z siłą eksplozji i dążyć do powrotu do swojej oryginalnej pozycji na modelu.



Rysunek 13 Ustawienia obiektów Nucleus i nCloth

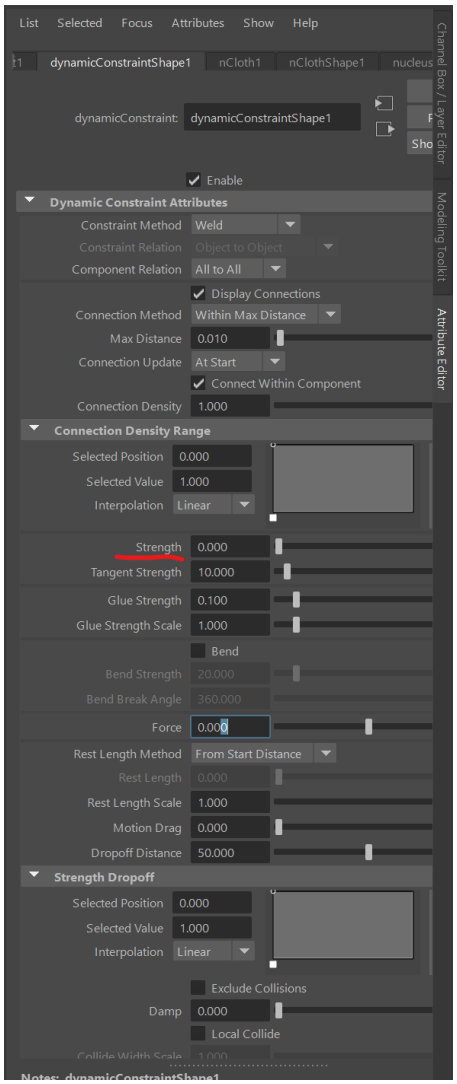


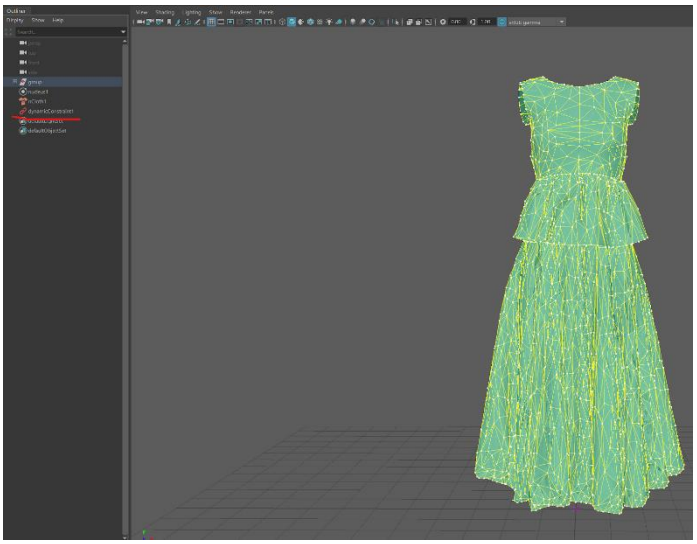
Wartości obu węzłów można dostosowywać w trakcie testowania działającej animacji. Teraz należy zadbać, żeby model sukienki faktycznie rozerwał się na małe fragmenty. W tym celu należy zaznaczyć odpowiednie miejsca do przecięć, ustawiając nCloth jako powierzchnię rozrywalną (*Tearable Surface*). W tym celu, zaznaczywszy obiekt, z menu górnego wybieramy rodzinę węzłów nConstraint, a typ węzła Tearable Surface.

Po chwili obliczeń widzimy, że wszystkie brzegi face'ów modelu zmieniły kolor, a ich przecięcia są zaznaczone kropkami. Oznacza to, że wszystkie zmieniły się na szwy do przerwania. Oczywiście można nie wybierać wszystkich brzegów, ale wybrać tylko te interesujące, zaznaczając te po kolei. Nie jest to jednak celem tego przykładu.

Po utworzeniu węzła widzimy, że pojawił się on w menu po lewej stronie. Wybierając go, mamy dostęp do jego właściwości fizycznych.

Rysunek 14 Ustawienie rozrywanej powierzchni na modelu

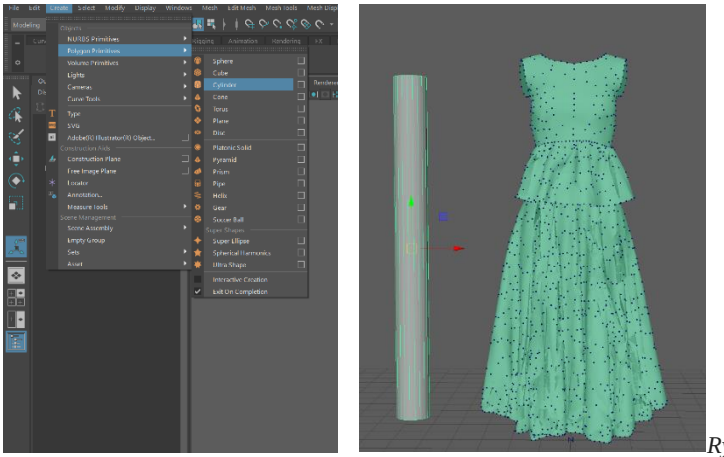




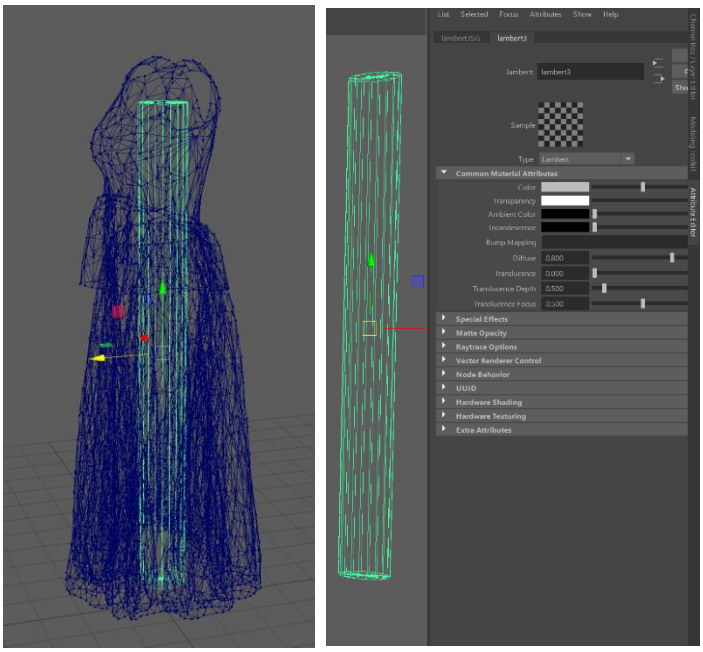
Rysunek 15 Model po dodaniu węzłów

Manipulując wartością siły (*Strength*) trzymania szwów można ułatwić lub utrudnić darcie się tkaniny. W tym przykładzie wartość jest ustawiona na 0, aby zapewnić, że prawie każdy fragment będzie łatwo odchodził od drugiego pod wpływem siły uderzenia.

Jeśli właściwości fizyczne materiału są już gotowe do rozerwania, należy teraz stworzyć siłę, która na ten materiał zadziała. W tym celu w menu pomocniczym należy przejść z FX na Modeling i stworzyć nową bryłę typu np. walec.



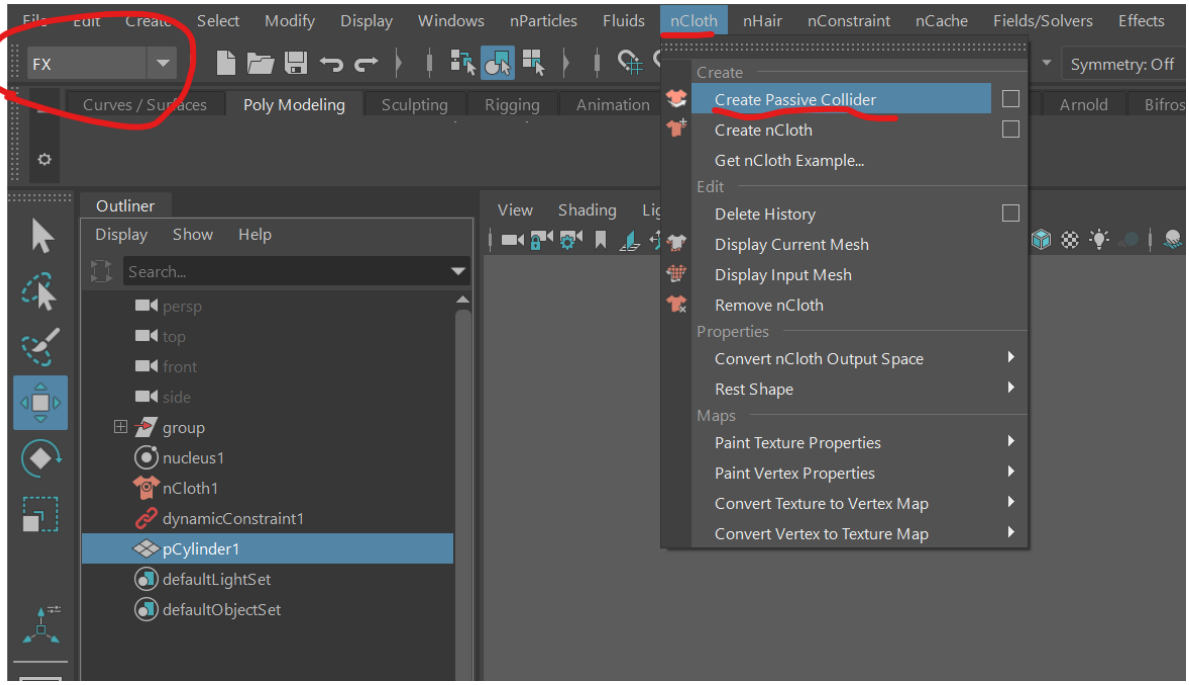
Rysunek 16 Wstawianie bryły



Korzystając ze strzałek do manipulacji wielkością bryły należy sprawić, by jej wysokość była zbliżona do wysokości modelu sukienki, natomiast obwód nie większy niż obwód modelu sukienki w najwęższym miejscu - chodzi o to, by bryły na siebie nie zachodziły, ale walec po przesunięciu znalazł się idealnie wewnątrz sukienki. Modelowanie wielkości bryły może się odbyć wewnątrz rozrywanego modelu albo obok, ważne, aby dobrze to porównać.

Następnie bryle należy nadać teksturę - najlepiej typu Lambert - której transparentność trzeba ustawić na sto procent. W finalnej animacji nie chcemy widzieć bryły walca, ale jedynie siłę, z jaką oddziałuje na model, który rozrywa.

Kiedy już tekstura bryły jest niewidoczna, a bryła umieszczona jest wewnątrz modelu, możemy wykonać ostatnie kroki w celu uzyskania animacji. Najpierw należy nadać bryle własności obiektu fizycznego (*Passive Collider*) z menu FX.



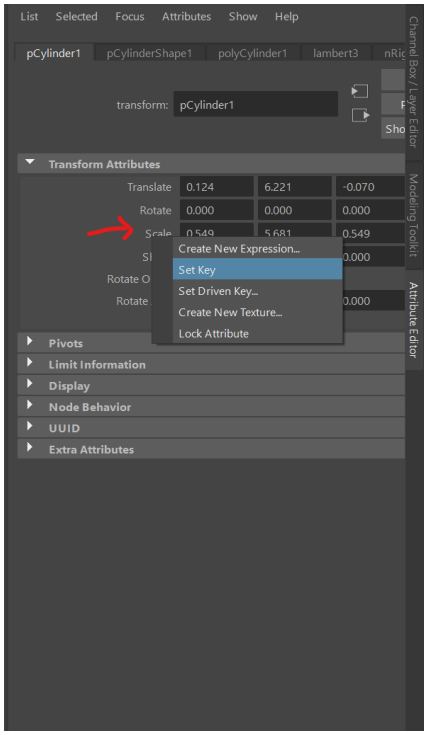
Rysunek 17 Utworzenie pasywnego obiektu

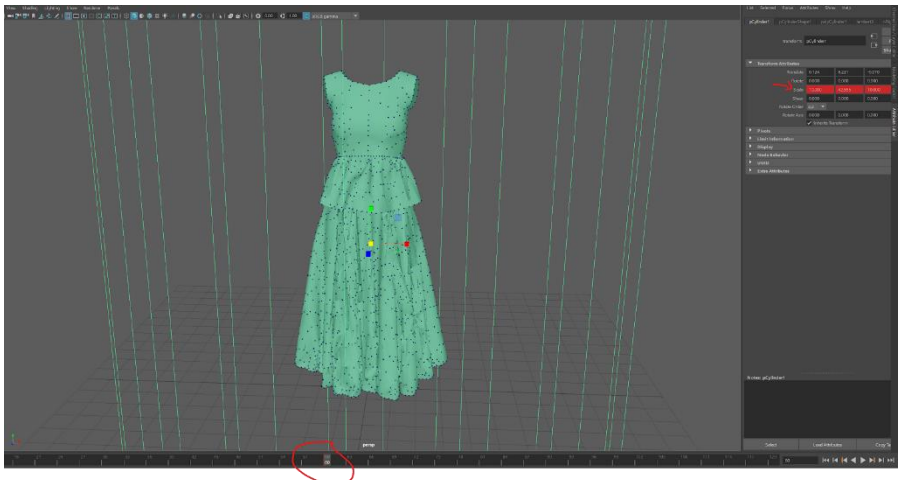
Następnie można przejść do samej animacji. Zatrzymując się na dowolnej klatce na osi czasu na dole okna programu należy zwrócić się do zakładki właściwości bryły walca. Jego obecną wielkość -

mieszczącą się w obrębie sukienki - chcemy zapisać jako początkową wartości animacji, Klikając prawym przyciskiem myszy na Scale należy wybrać opcję Set Key (ustaw klucz animacji):

Klucz zostanie ustawiony na danej klatce animacji na danych wartościach. Obszar Scale powinien podświetlić się na czerwono.

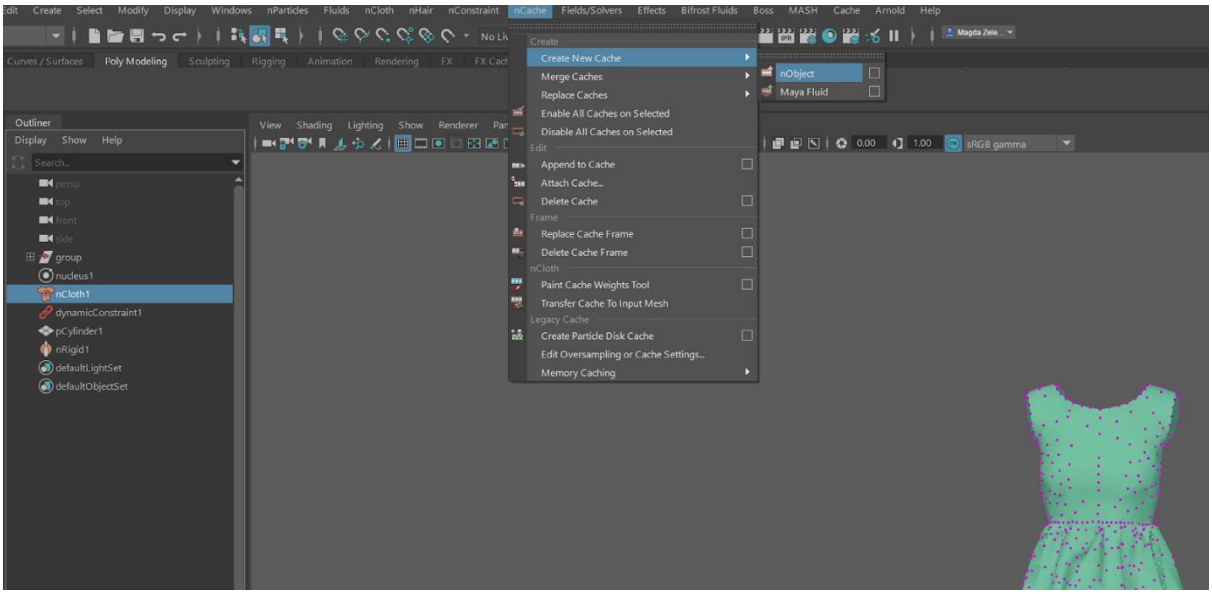
Następnie należy przesunąć się o kilkadziesiąt klatek dalej (tyle, ile chcemy, aby trwał nasz “wybuch”) i korzystając ze strzałek modyfikujących wielkość bryły lub edytując wartości x,y i z Scale zwiększyć walec tak, aby znacznie wychodził poza obszar sukienki. Następnie znów klikamy prawy przycisk na Scale i wybieramy Set Key.





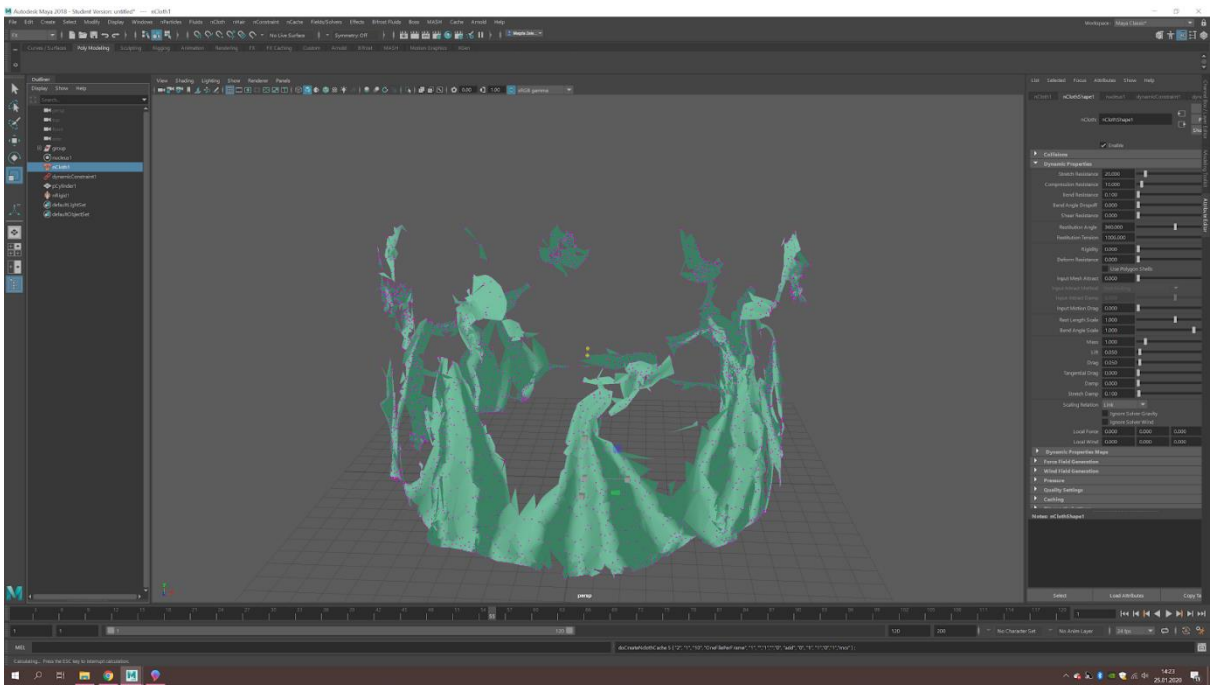
Rysunek 18 Ustawianie kluczy animacji

W ten sposób zostało przygotowane już wszystko, co niezbędne do wywołania podglądu animacji. Można teraz kliknąć play przy osi czasu, ale znacznie szybszy i optymalniejszy podgląd animacji uzyska się, wykonując wstępne obliczenia i zapisując je w pamięci tymczasowej. W tym celu wykorzystamy narzędzie nCache. Należy zaznaczyć obiekt nCloth na liście obiektów, a następnie z menu górnego FX wybrać nCache -> Create new Cache -> nObject. W ten sposób zapamięta się wartość obiektu nCloth we wszystkich klatkach animacji.



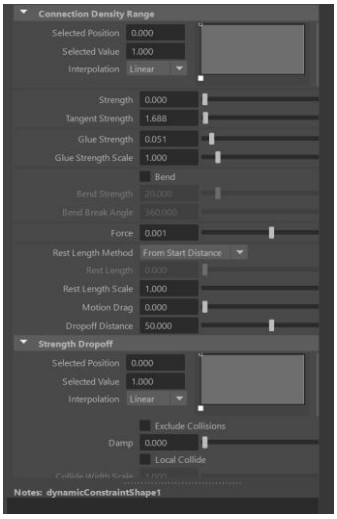
Rysunek 19 Pre-renderowanie animacji do pamięci tymczasowej cache

Teraz można chwilę poczekać na obliczenia. W skończonym czasie, dłuższym lub krótszym w zależności od skomplikowania modelu i wartości ustawień, nCache przeliczy całą animację i będzie można podejrzeć projekt w czasie rzeczywistym.



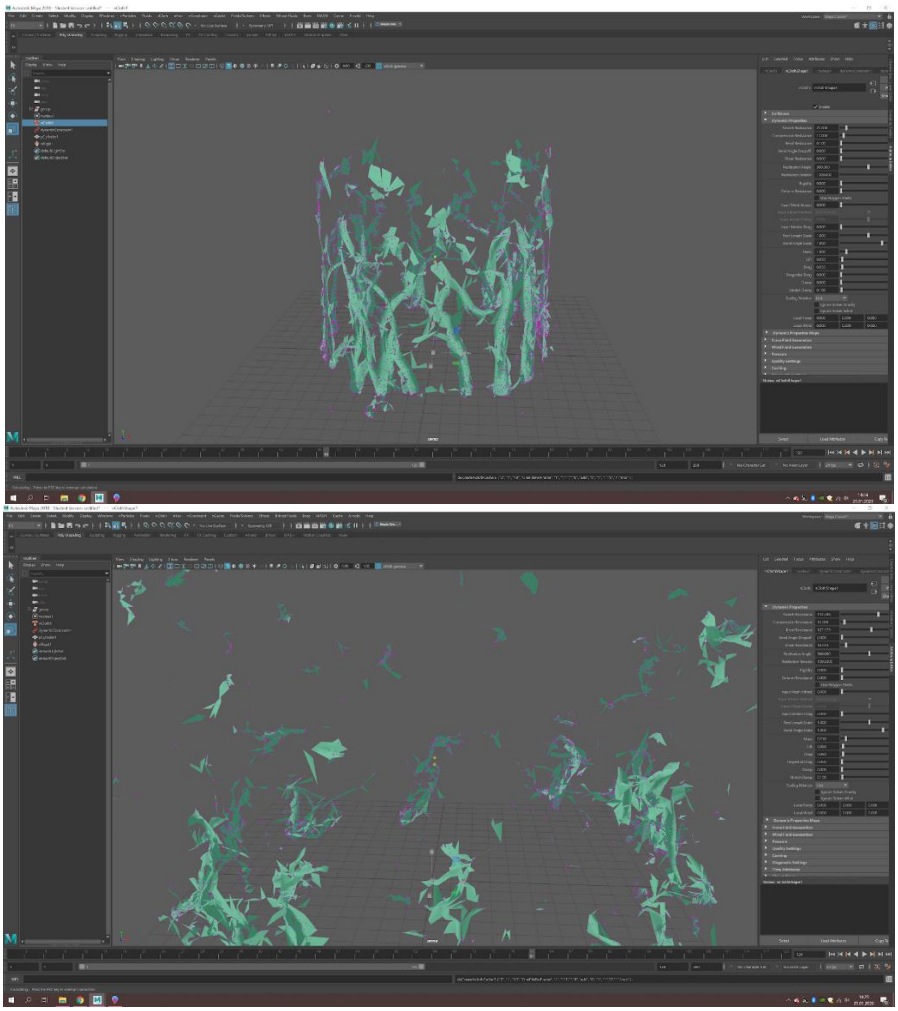
Rysunek 20 Efekt wybuchu modelu wg inicjalnych ustawień

Dzięki nCache możemy jeszcze w obrębie Maya zobaczyć dokładnie jaka jest dynamika i przebieg animacji oraz poprawić wartości przed renderingiem. Na tym przykładzie widać, że efekt nie został osiągnięty, materiał zostaje rozerwany na dosyć duże połacie, a chodziło o uzyskanie charakteru inspirowanego particles.



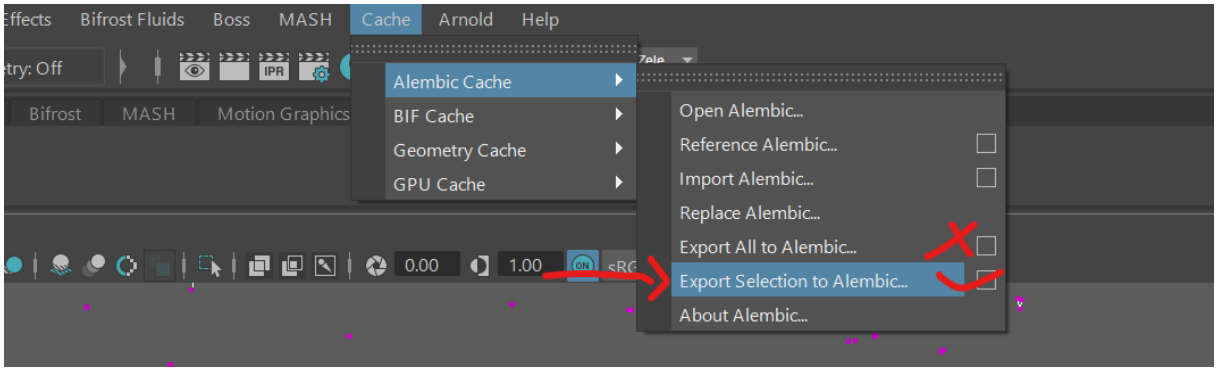
Rysunek 21 Zmiana ustawień Tearable Surface

W celu poprawy można pobawić się ustawieniami Strength i Bend Resistance nCloth i sprawdzić, co pozwala uzyskać najlepsze efekty. W tym przykładzie jednak zwiększam nieco wartość siły rozpadu szwów w komponencie nConstraint (Tearable Surface). Ta właściwość ma dużą moc, więc zwiększenie jej o 0.001 jednostki powinno już znacząco wpłynąć na końcowy efekt.



Rysunek 22 Efekt wybuchu po zmianie ustawień

Na powyższych obrazach widać, że jakość eksplozji znacznie się poprawiła, animacja również jest dynamiczniejsza i ma szerszy zakres, a elementy wirują szybciej. Dobierając odpowiednio wartości oraz kształt wewnętrznej bryły można uzyskać bardzo różne efekty w zależności od potrzeb.
W ten sposób uzyskałam animację eksplozji wszystkich modeli. Animacje wyeksportowałam z Maya w formacie .abc – Alembic, który jest najobszerniejszym rozszerzeniem do zapisu informacji o obiektach 3D. Animacje następnie połączyłam w całość w programie do edycji wideo Davinci Resolve, gdzie dodałam im również spójną barwę.



Rysunek 23 Eksport animacji do formatu Alembic

5.1.4 Animacja ruchu tkaniny

Kolejnym przykładem użycia nCloth w tym projekcie jest symulacja zachowania powierzchni tkaniny podczas ruchu nadanego obiektowi. Proces dążący do osiągnięcia tego celu przebiegał następująco:

1. Tworzę obiekt typu walec, ustawiam go prostopadle do siatki sukienki, niewiele ponad nią
2. Nadaję walcu właściwości obiektu typu *Passive Collider*
3. Zaznaczam na sukience wierzchołki znajdujące się najbliżej walca, z wciśniętym klawiszem *shift*
4. Trzymając wciśnięty klawisz *shift* zaznaczam również siatkę walca
5. Z menu FX wybieram utworzenie więzów między obiektami
6. Animuję w czasie poziomy ruch walca od punktu początkowego w klatce 1 do oddalonego w dowolnej klatce (hipotetycznie 200)
7. Jeśli siatka sukienki nie ma nałożonego symulatora nCloth, tworzę symulator nCloth
8. Uruchamiam animację

Po poprawnym wykonaniu tych czynności obiekt sukienki podąży przyczepiony do walca, a struktura sukienki będzie falować niczym przymocowana do wieszaka. Utworzona zostanie symulacja ubrania w ruchu.

5.1.5 Eksport animacji bez użycia silnika fizycznego

Three.js domyślnie nie obsługuje fizyki. Co za tym idzie, wykorzystanie animacji działających na podstawie fizyki jest niemożliwe bez użycia dodatkowych bibliotek z silnikiem fizycznym lub napisania

kodu własnoręcznie. W przypadku tego projektu, animacja jest zbyt prosta by dołączać pełen silnik fizyczny tylko do jej obsługi, lecz obiekt animowany na tyle skomplikowany, że pisanie kodu ręcznie nie wchodzi w grę.

Tę sytuację można jednak rozwiązać w inny sposób. Ponieważ fizyka w przeglądarce nie jest potrzebna ponad wykonanie samej animacji, można nagrać samą animację w programie 3D, a następnie w kodzie strony wykorzystać obiekt z już dołączoną animacją.

W standardzie możliwym do zapisu takiej animacji tkaniny formatem jest format alembic (.abc). Alembic zapisuje animację wraz z cache (pamięcią podręczną), w której przechowywane są informacje o właściwościach fizycznych, przemieszczeniu wierzchołków etc. Three.js jednak nie obsługuje formatu alembic ani cache.

Formatami, jakie obsługuje three.js, które są w stanie zapisać informacje o animacji, są FBX oraz GLTF. Żaden z nich jednak nie przechowa informacji o właściwościach fizycznych animacji bez wsparcia cache. Można jednak dokonać transformacji animacji materiału na deformację obiektu animowaną w czasie. Taki rodzaj informacji jest możliwy do zapisu i odtworzenia w Three.js.

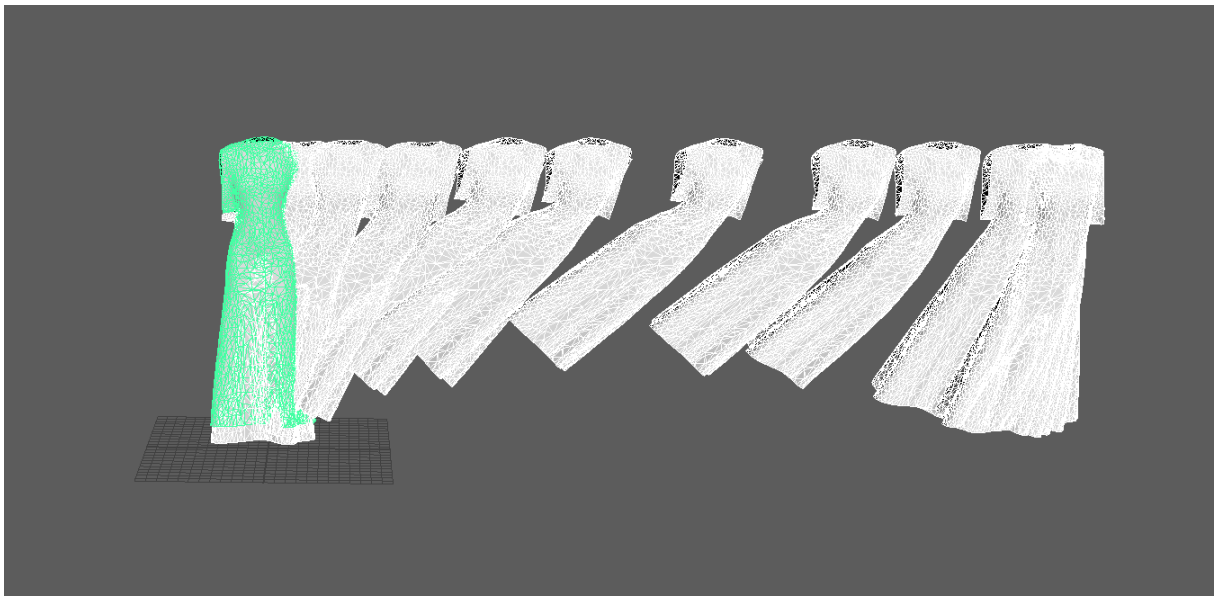
W tym miejscu warto rozszerzyć informacje o tym, czym są deformacje obiektów oraz jaka występuje w nim nomenklatura. Deformacja to przekształcenie bazowego obiektu w inny kształt lub inny obiekt. Podstawową zasadą, dzięki której przekształcenie w ogóle jest możliwe jest zachowanie identycznej topologii deformowanych obiektów. Rozumie się przez to, że obiekt podstawowy jak i ten, w który ten obiekt ma się zdeformować muszą mieć dokładnie taką samą liczbę płaszczyzn i wierzchołków. Za dokumentacją[13] Maya przy deformacjach obiektów mamy do czynienia z takimi pojęciami, jak:

1. Obiekt podstawowy (base object) - obiekt, który chcemy deformować
2. Kształt podstawowy (base shape) - pierwotny kształt obiektu przed deformację
3. Obiekty docelowe (target objects) - zdeformowane obiekty, które są najczęściej duplikatami obiektu podstawowego. Ten element jest opcjonalny przy deformacjach, gdyż tak naprawdę można jako docelowe kształty zapisać przekształcenia siatki podstawowej
4. Kształty docelowe (target shapes) - kształty obiektów docelowych lub kształty deformacji obiektu podstawowego
5. Kształt połączony (blend shape) - kształt powstały w wyniku nałożenia kształtu docelowego na obiekt podstawowy. Można modyfikować wagę przekształcenia, w skali od 0 do 1, gdzie 1 to pełen kształt docelowy.
6. Proces, w jakim została wykorzystana deformacja obiektu do zapisu animacji materiału jest opisany poniżej.

W pierwszym kroku należy odtworzyć animację materiału i duplikować wybrane kształty obiektu w czasie jej trwania. Po zduplikowaniu kształtu należy pozostawić go w tym samym miejscu względem

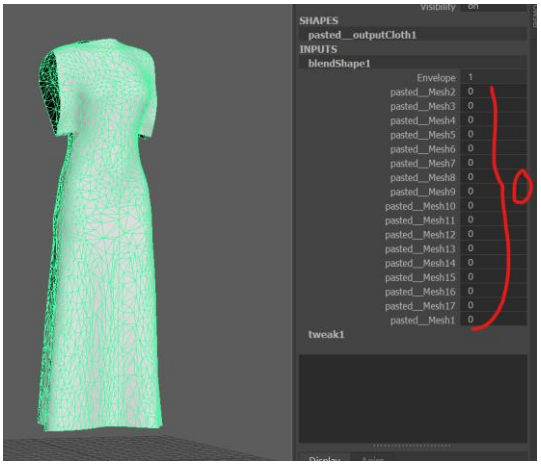
[13] Dokumentacja Maya, Blend Shapes…, https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2018/ENU/Maya-CharacterAnimation/files/GUID-01EFEC6D-41EA-46AA-81B1-C171DA4316F4-htm.html

pozostałych kształtów. Należy też usunąć historię obiektu, aby uzyskać tylko czysty zdeformowany model pozbawiony animacji czy wpływów fizycznych.



Rysunek 24 Zduplikowane kroki animacji modelu

Po zakończeniu tego procesu należy zaznaczyć wszystkie zduplikowane kształty i dla wygody przenieść je do osobnego pliku. Następnie wybrać z narzędzia Outliner lub wśród obiektów ten, od którego zaczynać ma się animacja. To jest obiekt i kształt podstawowy. Dobrze jest zmienić jego nazwę np. na "obiekt bazowy" lub wyróżnić w dowolny inny sposób.



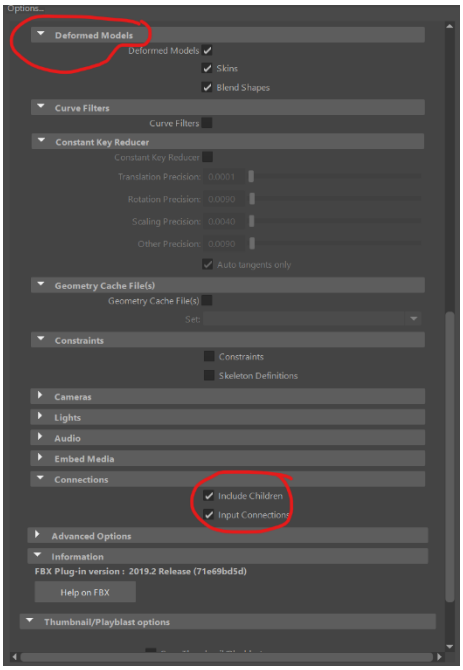
Rysunek 25 Narzędzie do animowania blend shapes

W kolejnym kroku należy zaznaczyć wszystkie obiekty poza kształtem bazowym, a następnie wciskając klawisz Shift dodać do zaznaczenia kształt bazowy. Posiadając zaznaczone kształty z menu należy wybrać Deform > BlendShapes. Na pierwszy rzut oka nic się nie wydarzyło, ale warto w tym momencie otworzyć właściwości Blend Shape dostępne w Attribute Editor. Można tutaj znaleźć odnaleźć "Obiekt bazowy" a pod nim listę z nazwami obiektów przekształceń, koło których znajdują się suwaki. Domyślnie wszystkie z nich mają wartości 0. Kiedy wszystkie kształty przekształceń mają wartość 0, obiekt podstawowy ma formę kształtu podstawowego. Jeśli zmienimy wartość np. pierwszego obiektu na liście na 1, obiekt podstawowy przyjmie kształt docelowy tego obiektu.

Tę operację można animować. Pozostawiając wszystkie wartości na 0, należy kliknąć prawym przyciskiem w blendShape1 i wybrać "Key all selectable". Doda to klucze animacji do wszystkich wartości, które potencjalnie mogłyby ulec zmianie. Bardziej optymalnym, ale nieco bardziej żmudnym rozwiązaniem jest dodawanie kluczy do tych wartości, które faktycznie ulegają lub ulegną zmianie w

następnej klatce. Po dodaniu kluczy na pierwszej klatce należy przesunąć się na następną, najlepiej oddaloną od pierwszej o sumę wszystkich klatek podzieloną przez liczbę wszystkich kształtów - lub dowolnie inną, byle oddaloną dla efektu o więcej niż 1. W tym miejscu należy zmienić wartość przy pierwszym zdeformowanym kształcie na 1 i zachować klucze. Następnie znowu się przesunąć na osi czasu, zmienić wartość tego kształtu na 0, a kolejnego na 1 i znów zachować klucze. Taką operację należy powtarzać aż do przestawienia wartości wszystkich kształtów.

Po zakończeniu animowania kształtów należy zaznaczyć obiekt bazowy oraz blendShape1 i z menu wybrać Deform->BlendShape->Bake Shapes to Targets. To bardzo istotna operacja do odtwarzania animacji w Three.js, gdyż kod biblioteki operuje właśnie na przeliczaniu kolejnych targetów przekształceń i nie interesuje go rzeczywisty wygląd siatki w danym momencie.



Rysunek 26 Ustawienia podczas eksportu do .fbx

Następnie należy wyeksportować plik do FBX. Eksport odbywa się w File->Export. Z właściwości FBX trzeba wybrać animation, bake animation, oraz zaznaczyć, by eksportowały się deformacje.

Plik FBX jest już plikiem normalnie do użycia w kodzie aplikacji. Dokumentacja Three.js zachęca jednak do korzystania z formatu GLTF[14], który jest znacznie lepiej wspierany przez tą bibliotekę. Osobiście zauważyłam też różnicę w rozmiarze pliku na korzyść GLTF. Dodatkowym atutem tego formatu jest obecność zaktualizowanej i działającej wersji loadera GLTF w bibliotece, podczas gdy aktualną działającą wersję loadera FBX musiałam wgrać ręcznie - co za tym idzie prawdopodobnie aktualizować ręcznie przez jakiś czas. Wadą GLTF jest domyślny materiał, który prezentuje się gorzej niż FBX, ale można go zmienić za pomocą shader'a.

Program Maya nie umożliwia eksportu plików do GLTF. Umożliwia to natomiast program Blender. Ponieważ lepiej znam Maya do modelowania i animacji, używam Blendera tylko do konwersji, niemniej sam ten program jest w stanie świetnie obsłużyć cały proces deformacji obiektów i animacji.

Eksport do GLTF przez Blender jest tak prosty jak zaimportowanie pliku FBX do nowej sceny w Blender, a następnie wyeksportowanie go do GLTF. Ten sam proces można odbyć bez użycia drugiego programu, korzystając z wtyczki przygotowanej przez twórców biblioteki Three.JS. Konwerter ten obsługiwany jest przez wiersz poleceń. Jego przewaga polega na możliwości konwersji całej puli plików na raz, a nie jednego na raz jak w przypadku Blendera.

[14] Dokumentacja Three.js, Animation System, https://threejs.org/docs/#manual/en/introduction/Animation-system

Ten opis konkluduje przygotowanie modeli ubrań do użycia podczas implementacji projektu oraz użycie symulatora nCloth. W dalszej części zostaną opisane inne metody praca z Maya wykorzystane podczas realizacji pracy.

5.1.6 Techniki generatywne

Oprócz symulacji fizycznych Maya oferuje znacznie więcej możliwości poza samym modelowaniem obiektów. W realizacji tego projektu zostały użyte również techniki generatywne, dokładnie metoda zwielokrotniania obiektów przy kopiowaniu ich instancji oraz metoda generowania cząstek (particles), już wspomniana we wcześniejszej partii tej pracy. Obie metody są szczegółowo opisane poniżej.

5.1.6.1 MASH

MASH jest narzędziem do generowania proceduralnego w Maya. Oznacza to, że na bazie jednej siatki (lub kilku) możemy wygenerować wiele, nawet bardzo skomplikowanych, sekwencji obiektów. Co jest istotne, narzędzie takie jak MASH nie kopiuje pierwotnego mesh w celu zwielokrotnienia, ale kopiuje jego instancję, czyli informację o jego właściwościach. Takie podejście znacznie optymalizuje pracę nad wieloma obiektami w jednej scenie. Dodatkowo, cały czas zachowany jest pierwotny obiekt, na podstawie którego wykonano sieć. Ten obiekt można podczas trwania całego procesu pracy modyfikować i animować, a nadane mu właściwości zostaną automatycznie skopiowane na resztę jego instancji.

MASH jest zaprojektowany na zasadzie sieci węzłów, z których każdy jest odpowiedzialny za inną specjalistyczną formę operacji na instancjach. Najczęściej używanym węzłem jest *Distribute*, którego funkcjonalność jest w zasadzie opisana powyżej – węzeł ten odpowiada za rozdystrybuowanie instancji na siatce według podanych mu wartości. Ilość instancji może iść w tysiące, a ich rozproszenie ustalić można na regularne na siatce, sferyczne, rozproszone… Liczba opcji i ich kombinacji jest duża.

Sieć MASH posiada 25 różnych rodzajów węzłów[15] do wyboru. Każdy z nich jest bardzo potężny i daje twórcy duże możliwości. Wymieniając kilka z nich dla przykładu:

- *Offset* – umożliwia przesunięcie, rotację czy skalowanie kolejnych instancji względem siebie
- *Color* – daje opcję randomizacji kolorów na materiale instancji
- *Time* – jest używane do animacji poszczególnych instancji według schematu

5.1.6.2 nParticles

Cząsteczki (*particles*) w ogólnym znaczeniu wykorzystuje się do symulacji ulotnych zjawisk, takich jak wybuchy, ogień, dym czy chmury. Sama cząsteczka jest albo punktem, albo dwuwymiarowym wielokątem albo trójwymiarową siatką. System cząsteczek (*particle system*) ma za zadanie imitować

[15] Dokumentacja Maya, Node Overview, https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2018/ENU/Maya-MotionGraphics/files/GUID-D4FECFDC-F91A-4BDC-A1B0-A24EB087B2DD-htm.html

zachowanie bezwładnych bytów wyrzuconych z jakąś siłą z punktu emisji (*emitter*). Zależnie od skomplikowania matematycznego systemu, cząsteczki mogą posiadać masę i tarcie i reagować na siebie wzajemnie, odbijając się od siebie czy opadając. System może również pomijać właściwości fizyczne samych cząstek za wyjątkiem nadanej im prędkości.

W Maya dostępny jest system nParticles, który, tłumacząc z dokumentacji w języku angielskim:

> (…) jest systemem generowania cząsteczek, który wykorzystuje Maya Nucleus, ten sam dynamiczny framework który generuje symulacje nCloth. Obiekt nParticle może kolidować z innymi obiektami nParticle, tak samo jak z obiektami nCloth czy pasywnymi przeszkodami(…).[16]

Niestety, system nParticles, podobnie jak nCloth jest obsługiwany w zamkniętym środowisku Maya i nie da się do wyeksportować jako animację. W przeciwieństwie do nCloth, utworzenie *Blend Shapes* również jest niemożliwe w tym przypadku. System nParticles w tym projekcie posłużył mi jedynie za inspirację do napisania własnego systemu już w kodzie strony, który to będzie opisany w dalszej części tekstu. Warto jednak przytoczyć nParticles opisując możliwości generatywne programu Maya.

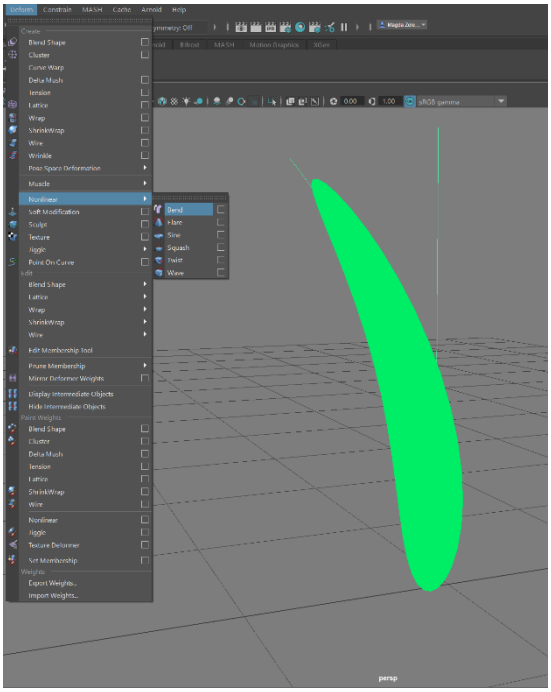
5.1.7 Animacja krajobrazu przy użyciu Maya MASH

W projekcie pojawia się scena komentująca używanie substancji chemicznych do zwiększania możliwości upraw plantacji bawełny. Zaprojektowałam tą scenę następująco: duża połać łąki kołysze się na wietrze w słońcu, bawełna wyrasta ponad wierzchołki traw. Po pewnym czasie trawa kurczy się i blaknie, a na scenie zostaje smutno kołysząca się samotna bawełna w bladym świetle.

Proces implementacji tego pomysłu w Maya MASH prezentują kolejne kroki:

1. Tworzę płaszczyznę. Dodatkowo modyfikuję ją wnosząc lub opuszczając niektóre ściany
2. Tworzę model pojedynczego źdźbła. W tym celu wybieram model sfery i rozciągam go w pionie oraz spłaszczam na grubości

[16] Dokumentacja Maya, nParticles introduction, https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2015/ENU/Maya/files/GUID-2AB4F791-4F12-4BCA-BA13-7A2F128FF3F4-htm.html

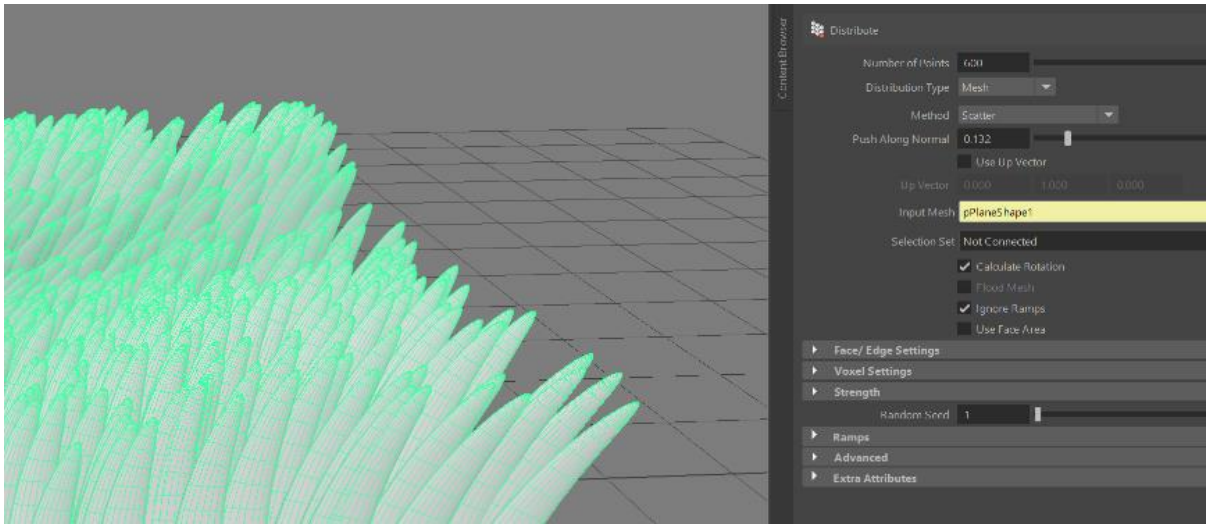


Rysunek 27 Deformacja źdźbła

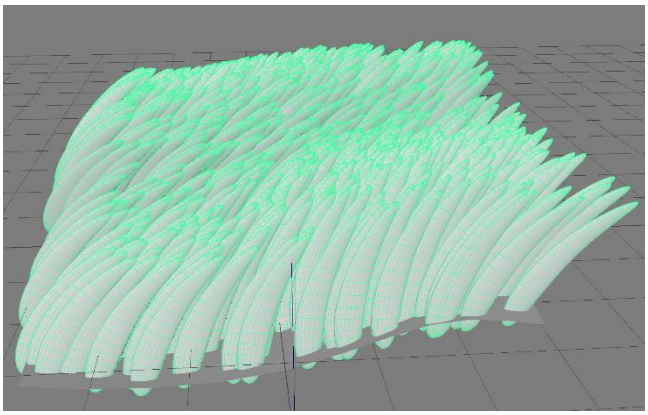
3. Tworzę animację pojedynczego źdźbła. W tym celu wybieram z menu Animation *Deform->Non-linear->Bend*. Do siatki źdźbła dodana zostaje krzywa, którą mogę dowolnie przekształcać, a cały mesh wygina się proporcjonalnie do niej (Rys. 27). Animuję wygięcia w czasie
4. Zaznaczając siatkę źdźbła z menu wybieram *MASH>Create MASH Network*. Do nowo utworzonej sieci jako pierwszy automatycznie dodaje się węzeł *Distribute*.
5. Manipuluję właściwościami węzła *Distribute* (Rys. 28):
   a. Ustawiam *Distribution Type* na mesh, gdyż chcę aby moje źdźbła ustawiły się na płaszczyźnie „ziemi”
   b. Jako *Input Mesh* ustawiam (przeciągając obiekt na pole input) utworzoną wcześniej płaszczynę
   c. Ustawiam metodę rozproszenia na *Scatter*, czyli najbardziej nieregularną, naturalną
   d. Zwiększam *Number of Points* na 600 dla poglądu
6. Dodaję kolejny węzeł, tym razem typu *Offset* – ustawiam jego *Transformation Space* na Local, gdyż chcę, aby mogę źdźbła przesuwały się względem swojej oryginalnej pozycji. Ustawiam *Strength* aż do uzyskania pożądanego efektu (Rys. 29)
7. Chcę, by źdźbła miały dodatkowo różne stopnie wygięcia. Dodaję węzeł *Random*, które ustawienia dotyczące rotacji czy skali pozwalają uzyskać zadowalający efekt (Rys. 30)
8. Źdźbła mają już animację nadaną pierwotnemu obiektowi, ale dla uzyskania imitacji naturalności dobrze jest wprowadzić element nieregularności również w „podmuchach wiatru”, które nimi falują. W tym celu dodaję węzeł *Time*, który posiada opcję *Stagger Frames* – umożliwia ona na przesunięcia stanu animacji w danej klatce różnych instancji
9. Na koniec pracy nad trawą dodaję węzeł *Color*, który pozwoli wprowadzić różnice kolorystyczne między poszczególnymi źdźbłami. Dzięki randomizacji odcienia i nasycenia efekt końcowy prezentuje się jak na rysunku 31
10. W osobnym pliku buduję model rośliny bawełny. Siatkę wraz z materiałem eksportuję jako Object 3D (.obj) w celu skompresowania modelu do najważniejszych informacji
11. Importuję mesh do sceny z trawą. Z siatką bawełny podstępuję jak wcześniej ze źdźbłem:
    a. Dodaję *Bend* i animuję go
    b. Tworzę dla niej **nowy** MASH Network, jednak przypinam *Distribution* do tej samej płaszczyzny

c. Ustawiam *Offset* i *Time* według uznania dla bawełny. Nie potrzebuję węzła *Random* ani *Color*

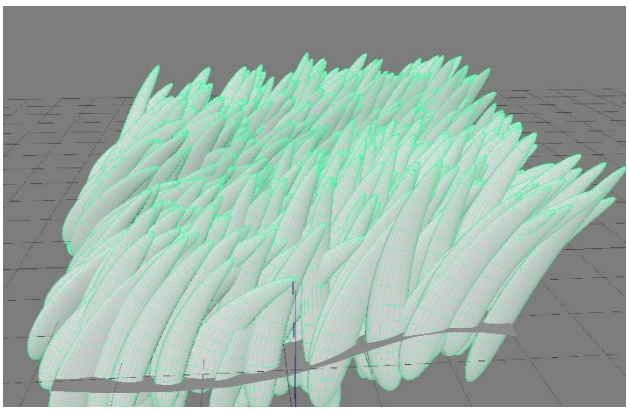
12. Dodaję płaszczyźnie gruntu teksturę spękanej ziemi. Tę samą teksturę dodaję również jako *Bump Map*, aby wzmocnić efekt dziur i bruzd
13. Dodaję do animacji trawy efekt zanikania. W czasie zwiększam *Offset* aby źdźbła zmniejszyły się i zapadły w grunt
14. Zmieniam wartość nasycenia koloru trawy w czasie, aby wyglądała na wyblakłą
15. Końcowy efekt renderuję jako sekwencję obrazów (Rys. 32)



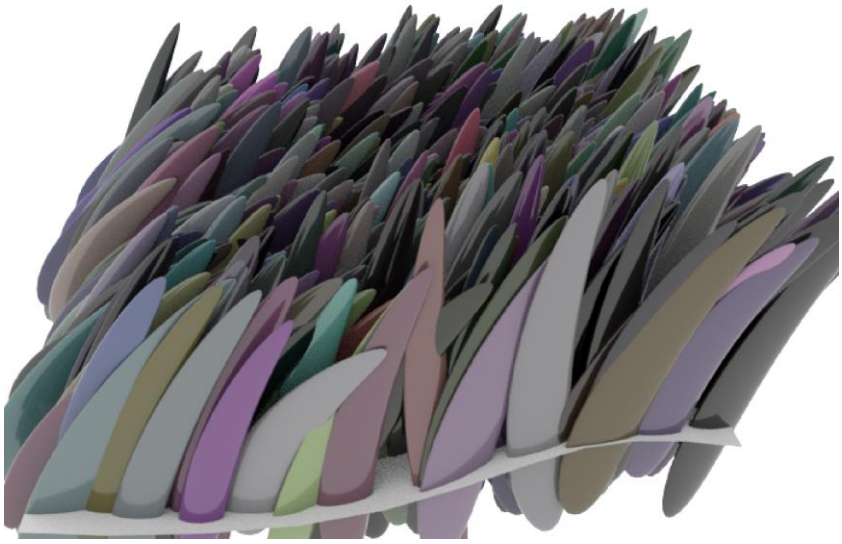
Rysunek 30 Węzeł rozproszenia



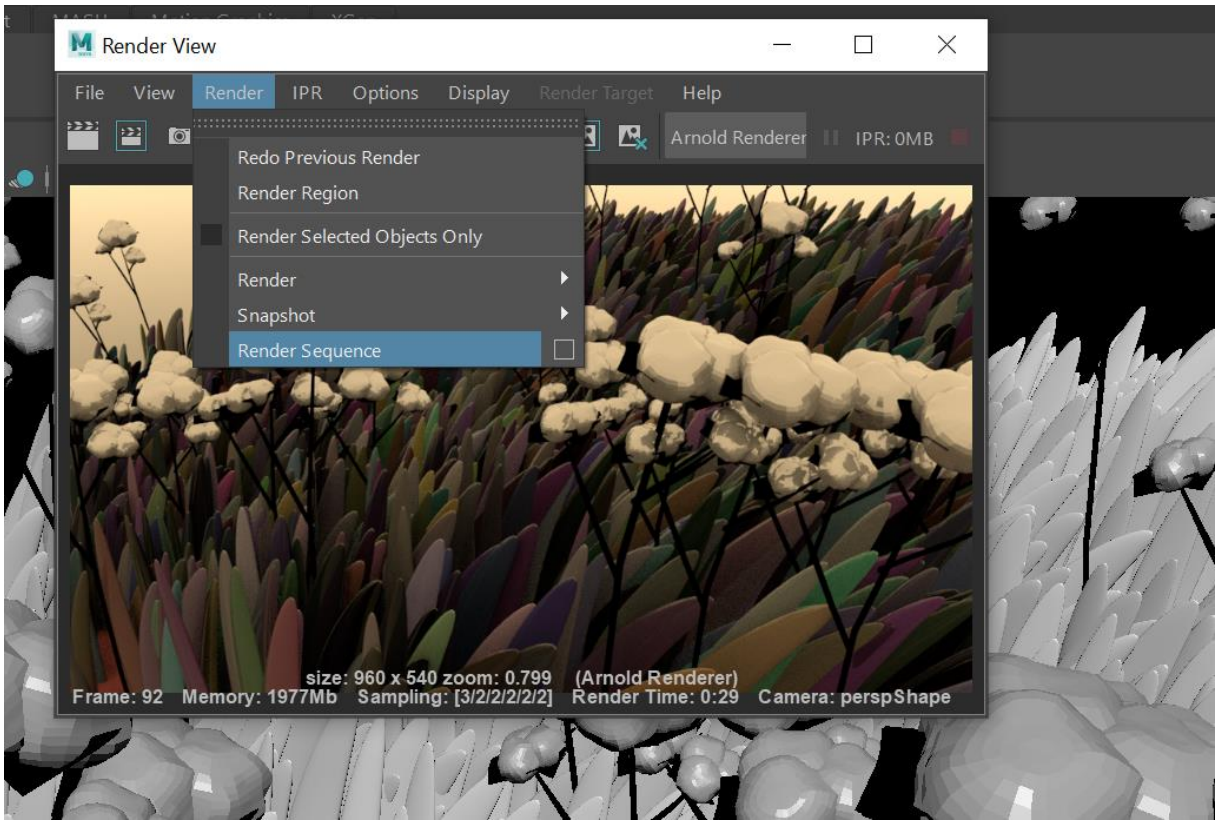
Rysunek 29 Zastosowanie Offset



Rysunek 28 Zastosowanie Random



Rysunek 31 Render trawy



Rysunek 32 Wybór renderu sekwencji na zakończonym projekcie MASH

Jako sekwencja poszczególne klatki animacji renderują się do osobnych plików graficznych. Aby zbudować z nich film, najlepiej jest posłużyć się dodatkowy programem do obróbki wideo. W tym celu użyłam znów programu Davinci Resolve, dzięki któremu wyeksportowałam swoją animację do pliku .mp4.

5.1.8 Podsumowanie

W tej sekcji opisałam wykorzystanie programów graficznych do wykonania materiałów 3D. Część z nich powstała jako kompletne, mogące być prezentowane niezależnie projekty, część jako półprodukty, z których będę dalej korzystać w pracy z kodem. W tej sekcji chciałam również wskazać, jak duże możliwości oferuje software 3D i jak ważna jest podstawowa wiedza fizyczna do pracy z animacją i symulatorami.

5.2 BUDOWA APLIKACJI WEBOWEJ

5.2.1 Wprowadzenie

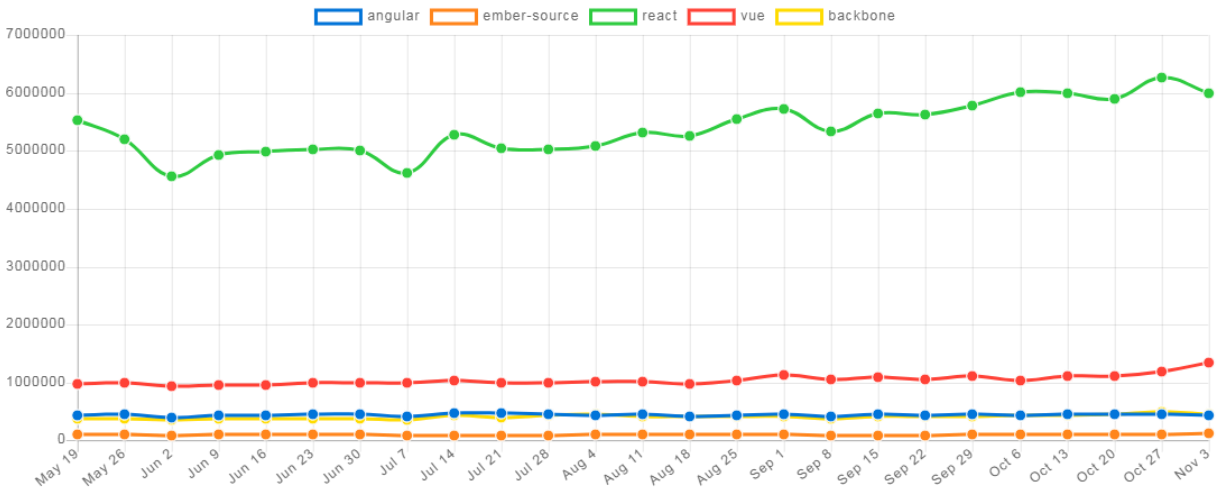
Tak jak było to wspomniane we wstępie, projekt jest w swoim założeniu stroną www bez dodatkowych podstron, na której treści pojawiają się w miarę przewijania. Produkt o takim charakterze można by uzyskać w „czystym" JavaScript, bez szczególnego dodatkowego wsparcia. Jednak projektując architekturę pracy uznałam, że warto rozważyć podejście SPA (ang. Single Page Application, aplikacja jednostronicowa). Strona nie zawiera cech aplikacji, nie przenoszone lub procesowane są żadne dane, użytkownik ma tylko jeden scenariusz wyboru do przejścia interfejsu, o ile można mówić o interfejsie – strona nie jest programem, jest wyłącznie wyświetlanym zestawem obrazu i tekstu. Rozważając podejście SPA myślałam jednak o i ilości danych do przeliczenia na witrynie w kontekście grafiki 3D. Stąd zrodził się pomysł zapewnienia dodatkowej kontroli do poszczególnych komponentów strony.

Takie podejście zapewnia biblioteka React.js, z którą miałam doświadczenie już wcześniej. React umożliwia łatwe kontrolowanie stanu poszczególnych komponentów i wyświetlanie (renderowanie) konkretnych informacji dopiero po spełnieniu określonych warunków. Co więcej, korzystając z React ułatwiona jest kontrola asynchroniczności w przypadku ładowanych obiektów zewnętrznych, co jest niezmiernie istotne, gdyż wiedziałam, że będę korzystać z ciężkich objętościowo materiałów, takich jak wideo.

W tej sekcji tekstu szczegółowo opisana jest architektura aplikacji, jak i również sposób przygotowania do pracy oraz instalacji poszczególnych zależności.

5.2.2 React.js

Przed rozpoczęciem opisu implementacji projektu, warto nakreślić charakterystykę biblioteki React.js. To bardzo szybka, wydajna biblioteka do tworzenia interfejsów użytkownika. Na rok 2020, kiedy powstaje ta praca, React pozostaje numerem jeden wśród twórców front-end:



Rysunek 33 Trendy w poborze paczek od 19 maja 2019 do 3 listopada 2019[17]

React charakteryzuje kilka elementów, dzięki którym jest tak łatwy w użyciu:

1. Jest to biblioteka, nie framework – to oznacza, że możemy ją zainstalować do już istniejącego projektu i przerobić według uznania. Nie potrzebuje dodatkowych zależności ani budowy specjalnego środowiska.
2. Ma modularną budowę – opiera się na komponentach, które można dowolnie łączyć i dzielić. Komponenty mogą być odpowiedzialne nawet za bardzo wąski obszar funkcjonalności, przez co mogą być re-używalne w dowolnym miejscu aplikacji lub nawet w innej.
3. Przechowuje w pamięci wirtualny obraz DOM – *document object model*, czyli struktury aplikacji webowej i podczas korzystania z interfejsu porównuje stany aplikacji i renderuje tylko zmiany – co ma znaczący wpływ na szybkość działania witryny.
4. Operuje na stanach i przekazywanych właściwościach komponentów, które można stale monitorować tak, aby odpowiadać natychmiast na ich zmiany.
5. Funkcjonuje w oparciu o Node.js, dzięki czemu można łatwo dodawać i eksportować zależności między komponentami albo instalować nowe pobrawszy je z sieci.
6. Do budowy widoku używa składni JSX, dzięki której elementów HTML nie trzeba podawać w łańcuchach znaków, ale przygotowywać równolegle z funkcjami JavaScript, które mają na nie wpływ.

5.2.3 Lista bazowych zależności

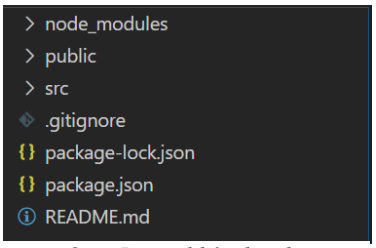
1. Webpack – to narzędzie do „pakowania" modułów, plików i zależności w statyczne, zbiorcze pliki do bezpośredniego serwowania klientowi.
2. Babel – biblioteka tłumacząca najnowszą składnię JavaScript (ten projekt korzysta ze składni ES6) na funkcje rozumiane przez wszystkie, również starsze, kompilatory przeglądarek

[17] Źródło: FreeCodeCamp, List of top JavaScript frameworks, https://www.freecodecamp.org/news/complete-guide-for-front-end-developers-javascript-frameworks-2019/

3. Sass – preprocesor języka CSS, który rozszerza jego składnię o możliwości do tej pory dostępne w językach programowania, takie jak funkcje, zmienne czy zagnieżdżanie modułów.
4. ESLint – narzędzie do monitorowania poprawności składni JavaScript
5. Zależności opisane we wstępie do pracy:
    a. Three.js
    b. GSAP
    c. Pixi.js
    d. ScrollMagic.js

5.2.4 Architektura aplikacji

Aplikacja jest skonstruowana zgodnie z zasadami pracy z React. W podstawowym układzie zawiera ona trzy foldery: folder z modułami node, folder *public* ze skompilowanym kodem produkcyjnym oraz folder *src* z plikami deweloperskimi. Dodatkowo pojawiają się dwa pliki json: plik *package* z listą wersji kodu, zależności i uruchamialnych skryptów oraz plik *package-lock*, który zawiera listę wersji zależności, które pozwalają na uruchomienie aplikacji. To ważne w przypadku współdzielenia repozytorium kodu, bo sam plik *package* może pozwolić na zainstalowanie nowszych zależności niż współpracujące z aplikacją. Ostatnim plikiem na głównej liście jest *readme*, powstałe na potrzeby repozytorium.



Rysunek 34 Lista plików katalogu głównego

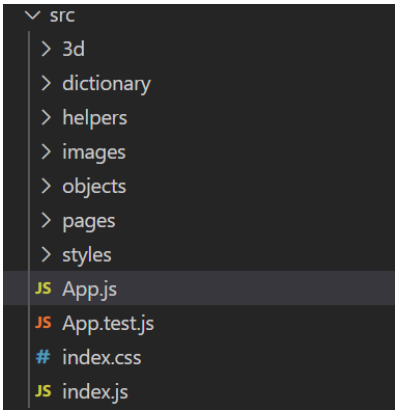
Folder wykorzystywany podczas pracy to folder *src*. W folderze znajduje się plik index.js, do które sprowadzane są wszystkie pozostałe. W pliku index.js w jako jedynym na całą aplikację znajduje się funkcja wywołująca renderer drzewa dokumentu:

```
1.  ReactDOM.render(
2.    <App />,
3.    document.getElementById('root')
4.  );
```

Element o id „root” znajduje się już w folderze *public*. Istotne w tym fragmencie jest odwołanie do komponentu App, który to łączy w sobie pozostałe komponenty użyte w aplikacji.

Struktura katalogu App jest następująca:

- Folder 3D zawiera obiekty 3D, skrypty do ładowania 3D oraz skrypty shader’ów
- Dictionary zawiera słowniki w formacie JSON
- Helpers funkcje pomocnicze wykorzystywane w wielu komponentach
- Objects reużywalne komponenty



Rysunek 35 Struktura katalogu src

- Pages poszczególne sekcje do wyświetlenia
- Styles style dokumentu

5.2.5 Komponent wejściowy jako przykład wykorzystania bibliotek animacji

Komponent, który uruchamia się jako pierwszy do uruchomieniu strony warto opisać jako przykład użycia bibliotek do animacji GSAP oraz ScrollMagic. W pozostałych komponentach są one używane analogicznie to tego przykładu.

Komponent renderowany jako pierwszy nosi nazwę LoadingApp. Oprócz pełnienia funkcji wprowadzającej do historii pod kątem treści, jest on również odpowiedzialny za upewnienie się, czy reszta materiału została poprawnie załadowana.

LoadingApp zaprojektowany jest w następujący sposób: najpierw animowany jest licznik progresu załadowania strony od 0 do 100. Gdy uzyska wartość 100 chwilowo zatrzymuje się, po czym zaczyna bardzo szybko rosnąć do 710 000 000. W chwili zatrzymania się na tej wartości, pojawia się tekst ponad i poniżej liczby, opisujący jej znaczenie. Napisy te animują się w ten sposób, że spoza ekranu pojawia się chmura liter, które układają się w słowa. Pojawia się też przycisk, po kliknięciu którego przechodzimy do kolejnej animacji. Liczba znów się zmienia, a napisy rozpadają się i składają znów w nowe.

Najłatwiej zrozumieć strukturę Jsx tego komponentu zaczynając od wyjaśnienia, jak dokładnie działa animacja napisu. Wszystkie teksty w aplikacji są przechowywane w słowniku, który importuje się na górze strony. Aby uzyskać efekt „rozsypania" słów, należy rozbić łańcuch znaków na poszczególne litery. W tym celu napisałam krótką funkcję, która mapuje każdą z liter na element typu *span* z tą literą wewnątrz:

```
1.  export var generateTextForAnimation = (text) => {
2.      return text.map((el,i) =>{
3.          return <span key={i}>{el}</span>;
4.      })
5.  }
```

W ten sposób, za każdym razem, kiedy do dokumentu dołączany jest tekst ze słownika, wystarczy uruchomić powyższą funkcję, aby wygenerować z niego zestaw elementów *span*. Wygenerowanie takiego zestawu zajmuje jednak kilka milisekund, więc musi zostać wykonane zanim podjęte zostaną jakiekolwiek próby animowania. W związku z tym najlepiej jest zawrzeć wszystkie teksty od razu w strukturze dokumentu, a potem dopiero decydować, który konkretnie wyświetlić w danym momencie.

Aby na bieżąco decydować, którą animację akurat należy wyświetlić, najlepiej skorzystać ze stanu komponentu. W obiekcie stanu można przechować dynamicznie zmieniającą się wartość. W ten sposób, dla trzech zaprojektowanych scen, można wywołać animację 1, 2 lub 3. W związku z tym przykładowo dla nagłówka (tekst powyżej licznika) struktura JSX wygląda następująco:

```
1.  <div className="introHeadline"
2.  ref= {e => this.headingTopRef = e}>
3.     { this.state.animation === 1 && (generateTextForAnimation(texts.pageOne.head-
    line.split('')))}
4.     { this.state.animation === 2 && (generateTextForAnimation(texts.pageTwo.head-
    line.split('')))}
5.     { this.state.animation === 3 && (generateTextForAnimation(texts.pageThree.head-
    line.split('')))}
6.   </div>
```

Dzięki temu, React wyrenderuje tekst nr 2 na stronie dopiero kiedy stan wskaże, że animacja ma również nr 2. Jednak funkcja *generateTextForAnimation* już na samym początku wygeneruje elementy *span*, a React tylko przechowa to w pamięci.

Posiadając kontrolę nad kolejnością animacji poprzez stan oraz mając gotowe zbiory *span*, można wykonać faktyczne animacje. W tym miejscu będzie można wytłumaczyć, jak działa GSAP. Do animowania tekstu wykorzystałam zmodyfikowaną funkcję według przykładu Nate'a Wiley[18]:

```
1.  import {Expo, TimelineMax} from 'gsap/all';
2.  export var animateText = (span, i) =>{
3.      let textAnimation = new TimelineMax()
4.      .from(span, 1, {
5.          opacity: 0,
6.          ease: Expo.easeIn,
7.          x: random(-50, 50),
8.          y: random(-5, 500),
9.          z: random(-50, 50),
10.         scale: .1,
11.         delay: i * .02
12.     });
13.     return textAnimation;
14. }
```

Opisując powyższy kod według kolejności linii:

1. Importuję TimelineMax oraz funkcję Expo z biblioteki GSAP:
    a. TimelineMax pozwala na ułożenie animacji jedna po drugiej według kolejności kodu. Pozwala na ustalenie czasu trwania danej funkcji oraz ewentualnego opóźnienia
    b. Expo jest funkcją opisującą *easing*, czyli wygładzenie progresu animacji. Expo zapewnia, że prędkość zmian w animacji będzie zachodziła eksponencjalnie, tj. najpierw wolno, potem coraz szybciej
2. Tworzę funkcję, która za argument będzie brała element typu *span* z literką oraz jego index w iteracji po zestawie
3. Tworzę tymczasowy element TimelineMax
4. Metoda *from* określa, że animacja będzie wykonywała od stanu opisanego w argumencie metody do stanu opisanego w stylu dokumentu. Styl elementów *span* w dokumencie zostanie pokazany poniżej. Czas trwania animacji (argument drugi) to jedna sekunda **dla każdego elementu *span***

[18] Wiley, Nate, Text animation using GSAP, https://codepen.io/natewiley/pen/xGyZXp

5. Wejściowa transparencja elementu jest 0 (element niewidoczny)
6. *Easing* animacji ustawiony na Expo (punkt 1.b)
7. W liniach 7 do 9 ustawiona jest wejściowa wartość pozycji translacji elementu według jego punktu 0. Translację wykonuje się za pomocą metody *transform* w CSS (przykład poniżej)
10. Skala jest również przedmiotem *transform*
11. Opóźnienie animacji litery względem indeksu iteracji

Posiadając gotową funkcję można ją wywołać w odpowiednim momencie przy użyciu ScrollMagic. Ta biblioteka umożliwia sprawdzenie odległości przewinięcia strony oraz kierunku. Mając te informacje, można precyzyjnie kontrolować wykonywanie animacji.

ScrollMagic składa się z dwóch filarów: kontrolera i sceny. Kontroler jest jeden na stronę i zawiera zbiór scen z informacjami, kiedy chcą być pokazane. Kontroler sprawdza informacje o przewinięciu strony i uruchamia odpowiednie sceny. W tej aplikacji jest jeden kontroler utworzony w App.js, przekazywany do komponentów przez dziedziczone właściwości. Warto przyjrzeć się konstrukcji sceny pierwszej, w której animowany jest pierwszy z tekstów, by wskazać cechy charakterystyczne jej budowy oraz wywołania funkcji animującej:

```
1.  this.scene1 = new ScrollMagic.Scene({
2.              duration: 50
3.          })
4.          .on('leave', () => {
5.       let requests = [...this.headingRef.getElementsByTagName('span')].map(item =>{
6.                  return new Promise(resolve => {
7.                      animateText(item, resolve).reverse(0);
8.                  })
9.              })
10.             Promise.all(requests);
11.            TweenMax.to(this.counterRef, 1, {
12.                fontSize: 200
13.            })
14.         })
15.         .on('enter', event =>{
16.             if(event.scrollDirection === "REVERSE"){
17.
18.                 this.setState({
19.                     counter: "710 000 000",
20.                     animation: 1
21.                 }, ()=>{
22.                     [...this.heading-
    Ref.getElementsByTagName('span')].fo-
    rEach((span, i)=>{
23.                         reanimateText(span);
24.                     });
25.
26.                 })
27.                 TweenMax.to(this.counterRef, 1, {
28.                     fontSize: 50
29.                 })
30.             }
31.
32.         })
```

1. Tworzę nowy obiekt sceny przypisany do zmiennej scene1

2. Ustalam czas trwania sceny na 50px. To niewiele, ale chcę by użytkownik po lekkim ruchu myszką czy pojedynczym naciśnięciu strzałki przeszedł do kolejnej
4. Funkcja zostanie wykonana na wyjściu ze sceny
5. Linie 5-10 zawierają wykonanie animacji tekstu przy użyciu Promise. JavaScript działa synchronicznie, co oznacza, że funkcje uruchomione w tym samym momencie wykonają się jedna po drugiej. Aby zapewnić, że funkcja *animateText* uruchomi się na wszystkich elementach *span*, użyty tutaj jest Promise, który sprawdza czy wpisana weń funkcja się wykona do końca na wszystkich elementach. Widać też tutaj, że *animateText* jako pierwszy argument przyjmuje element *span*, a na drugi index wykonywanego Promise.
11. Dodatkowo na liczniku wykonywana jest animacja wielkości tekstu przy użyciu TweenMax od GSAP. TweenMax jest zbliżoną funkcją do TimelineMax, jednak nie obsługuje zdarzeń wykonywanych po sobie, lecz pojedyncze.
15. W tej linii rozpoczyna się wykonanie zdarzenia na wejście w scenę, przy czym zaraz poniżej sprawdzany jest kierunek. REVERSE oznacza, że zdarzenie będzie obsługiwane na wejściu w scenę kierując się z dołu do góry.

Aby zamknąć ten wątek, pozostaje jeszcze pokazać, jaki jest inicjalny styl elementu *span*, do którego dąży animacja:

```
1.  .loadingSection{
2.      height: 100vh;
3.      width: 100%;
4.      position: relative;
5.      .introText, .introHeadline{
6.          position: fixed;
7.          text-align: center;
8.          font-size: 2em;
9.          span{
10.             display: inline-block;
11.             position: relative;
12.             min-width: .4em;
13.             transform: translate3d(0px, 0px, 0px) scale(1, 1);
14.         }
15.     }
16.     .introText{
17.         width: 80%;
18.         margin-left: 10%;
19.         bottom: 20vh;
20.     }
21.     .introHeadline{
22.         position: fixed;
23.         width:100%;
24.         margin-left: 0;
25.         top: 10vh;
26.     }
27. }
```

W liniach 9-14 opisane są tutaj style pojedynczego elementu *span*. Istotna jest zwłaszcza linia 13, w której ustawione jest domyślne *transform*. Translacja ustawiona jest na 0 (inicjalna pozycja) a skala

x i y na 1. To właśnie do tych właściwości wróci każdy element *span* po wykonaniu się funkcji *animateText*.

Powyższy przykład pokazał animację pierwszego nagłówka, ale dla pozostałych działa to analogicznie. Podobnie jest w przypadku liczników, chociaż one nie są pobierane z JSON, ale ze stanu aplikacji, gdzie zmieniają się zależnie od sceny. Funkcja animacji uruchamia się na *callback*, czyli zawołanie zwrotne funkcji zmiany stanu po jej zakończeniu. Przykładowo:

```
1.  this.setState({
2.      counter: "73%",
3.      animation: 2
4.      },
5.       ()=>{
6.       [...this.headingTopRef.getElementsByTagName('span')].forEach((span, i)=>{
7.                              animateText(span, i).play();
8.        });
9.        [...this.loader2Ref.getElementsByTagName('span')].forEach((span, i)=>{
10.                             animateText(span, i).play();
11.       });
12.     });
```

W linii 5 wywoływany jest *callback* funkcji *setState*, która zmienia wartość licznika na „73%” w drugiej animacji.

Ta sama logika dotycząca animacji pojawia się we wszystkich komponentach, nie zależnie czy tak jak tutaj operują tylko na tekście i liczbach, czy również na skomplikowanych operacjach 3D. Wykorzystanie podobnego schematu po pierwsze jest podyktowane API bibliotek, ale również ujednolica pracę w poszczególnych komponentach, które koniec końców zbierane są we wspólną całość w App.js.

5.2.6 Podsumowanie

React.js oraz inne użyte w projekcie biblioteki są rozbudowanymi narzędziami, którym można by poświęcić znacznie obszerniejsze opisy. Nie jest to jednak przedmiotem tej pracy, a wskazanie, jak wykorzystać można stosunkowo prosto te narzędzia w celu budowania nietypowych, efektownych interfejsów użytkownika.

5.3 GRAFIKA 3D W PRZEGLĄDARCE WWW

5.3.1 Wprowadzenie

W powyższych sekcjach zostało opisane przygotowanie prefabrykatów do pracy z 3D oraz implementacja środowiska, w których sceny 3D można tworzyć przy pomocy kodu. W tej sekcji przedstawię na własnych przykładach, jak budować skomplikowane sceny trójwymiarowe w kodzie aplikacji React. Zacznę od krótkiej charakterystyki standardu WebGL oraz biblioteki Three.js, a następnie opiszę implementacje konkretnych pomysłów przy ich użyciu.

5.3.2 WebGL

WebGL określa się mianem standardu, gdyż funkcjonuje jako standardowe API do tworzenia grafiki trójwymiarowej w przeglądarkach, choć nie jest oficjalnie objęte rekomendacjami W3C[19]. Projektem WebGL zarządza grupa Khronos, w skład której oryginalnie wchodzili po godzinach pracy wolontariusze z wiodących firm IT, takich jak Opera, Apple czy Google[20]. Na stronie grupy widnieje oficjalny opis czym jest WebGL:

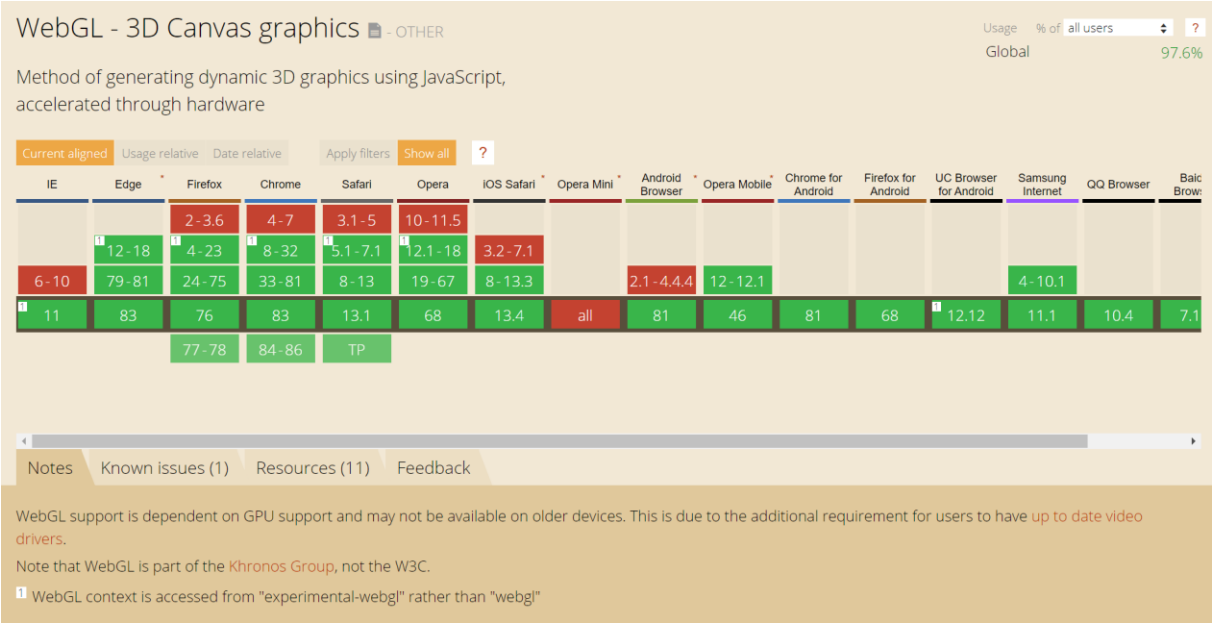
> WebGL to międzyplatformowy, darmowy standard webowy dla niskopoziomowego API grafiki 3D bazującego na OpenGL ES, wystawiony na ECMAScript poprzez element Canvas HTML5. Deweloperzy zaznajomieni z OpenGL ES 2.0 uznają WebGL za API bazujące na Shader'ach przy użyciu GLSL (…). WebGL umożliwia używanie 3D w sieci bez wtyczek, zaimplementowane bezpośrednio w przeglądarce. Główni dostawcy przeglądarek Apple (Safari), Google (Chrome), Microsoft (Edge) oraz Mozilla (Firefox) są członkami WebGL Working Group (grupy opracowującej WebGL).[21]

Kilka fragmentów tego opisu wymaga dodatkowego wyjaśnienia:

- OpenGL to standard renderowaniu grafiki trójwymiarowej na urządzeniu. Dzięki korzystaniu z OpenGL WebGL ma między innymi możliwość bycia wspomaganym przez silnik karty graficznej (przeglądarki są obsługiwane głównie przez CPU)
- ECMAScript to nazwa obowiązującego standardu JavaScript. Korzystanie z niego zapewnia WebGL elastyczność i obniża próg wejścia w naukę dla osób zaznajomionych z programowaniem aplikacji webowych
- Canvas HTML5 to element HTML5, który służy do generowania obrazów w przeglądarce przy użyciu JavaScript
- GLSL – Graphic Library Scripting Language – to osobny język do pisania skryptów shader'ów, który obsługiwać może bezpośrednio GPU

Chociaż w opisie powyższym twórcy chwalą się współpracą pośród dostawców wiodących przeglądarek, WebGL nie ma pełnego wsparcia w sieci. O ile na poniższym zestawieniu wsparcia dla WebGL większość wersji przeglądarek jest pokazana jako wspierana, WebGL nadal może nie działać w pełni na starszych urządzeniach, które nie wspierają akceleracji sprzętowej dla przeglądarek.

[19] Parisi, Tony, Aplikacje 3D, Helion 2015, str. 31
[20] Ibidem., str. 32
[21] Khronos Group, WebGL Overview, https://www.khronos.org/webgl/, tłumaczenie własne



Rysunek 36 Zestawienie wsparcia dla WebGL ze strony caniuse.com[22]

Standard WebGL jest bardzo potężny, ale dosyć niskopoziomowy. To oznacza, że praca z nim zakłada posiadanie mocno rozwiniętych zdolności matematycznych u programisty, gdyż grafikę 3D w WebGL opisuje się przy pomocy macierzy zapisywanych w postaci typowanych tablic. Nie istnieją predefiniowane funkcje pozwalające na budowanie skomplikowanych brył czy wzorów – wszystko musi zostać opisane po kolei na bazie współrzędnych wierzchołów i buforów przeliczania tych wierzchołków. Stąd praca bezpośrednio w WebGL jest jak najbardziej możliwa, ale dość uciążliwa i skutkuje powstawianiem bardzo obszernych programów do uzyskania niewielkich efektów. Dlatego właśnie korzystając z WebGL należy go rozumieć, ale zdecydowanie wskazane jest ułatwienie sobie pracy przy wsparciu bibliotek. Takimi bibliotekami są użyte w tym projekcie Three.js i PIXI.js.

5.3.3 Shadery i GLSL

W tej implementacji własne shadery są często wykorzystywane i zasługują na szczegółowe wyjaśnienie. W niektórych źródłach znaleźć można nazwę *shader* tłumaczoną na „cieniowanie”, ale może to być mylące, gdyż shadery wykorzystuje się w znacznie szerszym spektrum i w obecnym momencie słowo *shader* jest używane jako nazwa standardu, a nie konkretna funkcjonalność. Oryginalnie jednak faktycznie shadery służyły do cieniowania brył i nadal służą, są wpisane w standard takich materiałów jak Phonga czy Lamberta.

Specyfikę i genezę shaderów dobrze opisuje Tony Parisi w swojej książce *Aplikacje 3D*:

> W miarę ewolucji grafiki komputerowej oraz obserwowanego od 20 lat wzrostu wartości produkcji – początkowo dotyczącej filmowych efektów specjalnych, a później także dla gier wideo – cieniowanie przestało być tylko artystycznym zajęciem i stało się ogólnym problemem programistycznym. Zamiast prób przewidywania każdej możliwej kombinacji właściwości

[22] Źródło: https://caniuse.com/#search=webgl

> materiałów i kodowania ich w silniku wykonawczym, specjaliści z branży opracowali programowalną technologię zwaną programowalnymi shaderami albo w skrócie po prostu shaderami. Shadery umożliwiają pisanie kodu implementującego skomplikowane efekty dla pojedynczych wierzchołków i pikseli przy użyciu kompilowanego języka podobnego do C i wykonywanego przez GPU. Za ich pomocą można tworzyć bardzo realistyczne i efektywne grafiki, wolne od ograniczeń wiążących się z używaniem wcześniej zdefiniowanych materiałów i modeli oświetlenia.[23]

Rozszerzając ten opis, standard OpenGL na którym oparty jest WebGL posiada własny język programowania shaderów GLSL. Jak już zostało wspomniane to wcześniej, OpenGL działa przy wparciu GPU, czyli silnika karty graficznej, dzięki którym może korzystać z jej przetwarzania równoległego obliczeń na wielu rdzeniach. Sprawia to, że tekstury opisane przy użyciu shaderów są wyjątkowo wydajne.

W przypadku shaderów, które wykorzystane są w tej pracy, należy rozróżnić dwie podgrupy:

1. Vertex shader (*shader wierzchołków*) – odpowiada za transformację położenia wierzchołka z wirtualnej przestrzeni 3D na 2D wyświetlaną na ekranie. Modyfikowane może być przede wszystkim położenie, ale również kolor lub tekstura wierzchołków
2. Fragment shader (*shader fragmentów, pikseli*) – odpowiada za przeliczanie wartości i kolorowanie pikseli. Często odpowiedzialny za światło lub modyfikacje tekstury cieniem bez przesuwania wierzchołków

Na zakończenie opisu shaderów warto jeszcze wspomnieć, że chociaż są osobnymi bytami programistycznymi pisanymi w innym języku niż Three.js, podczas tworzenia scen 3D w tej bibliotece jest możliwa komunikacja między shaderem a sceną. Zostanie to pokazane na przykładach.

5.3.4 Tworzenie sceny 3D przy użyciu Three.js

Aby zbudować scenę 3D w środowisku webowym, potrzebujemy kontekstu Canvas HTML5 (tak, jak to podano w opisie WebGL).

```
1.  <div id="dressesSequence" ref={this.onSectionLoad}>
2.      <canvas  ref={ref=>this.canvasRef = ref}></canvas>
3.  </div>
```

Element *canvas* można utworzyć też bezpośrednio w kodzie JavaScript, ale dla porządku warto pokazać początkową strukturę takiej strony.

```
1.  const canvas = this.canvasRef.current;
2.          //scene
3.  this.scene = new THREE.Scene();
4.  this.scene.fog = new THREE.Fog(0x000000, 80, 100);
5.          //renderer
6.  this.renderer = new THREE.WebGLRenderer({ canvas, antialias: true, alpha: true });
7.  this.renderer.shadowMap.enabled = true;
```

[23] Parisi, Tony, *Aplikacje 3D*, Helion 2015, str. 91

```
8.  this.state.sectionRef.replaceChild(this.renderer.domElement, this.state.section-
    Ref.getElementsByTagName('canvas')[0]);
9.          //camera
10. this.camera = new THREE.PerspectiveCamera(
11.             50,
12.             window.innerWidth / window.innerHeight,
13.             0.1,
14.             1000
15.           );
16. this.camera.position.z = 30
17. this.camera.position.x = 0;
18. this.camera.position.y = -3;
```

1. Tworzę odwołanie do obiektu *canvas* w DOM
3. Tworzę scenę 3D
6. Tworzę renderer na bazie *canvas*
10. Dodaję kamerę do obserwacji sceny 3D

Kamera i scena muszą być odświeżane przez renderer w każdej klatce animacji (nawet, jeśli mamy do czynienia z obiektem statycznym). W tym celu wywołuje się je w specjalnej funkcji rekursywnej, która uruchamiana jest systematycznie wg metody *requestAnimationFrame:*

> Metoda Window.requestAnimationFrame() informuje przeglądarkę o zamiarze wykonania animacji i żąda od przeglądarki wywołania określonej funkcji w celu odświeżenia animacji przed następnym odmalowaniem. Argumentem metody jest funkcja (callback) do wywołania przed następnym odmalowaniem (odświeżeniem kanwy).[24]

To bardzo wydajna metoda, która upewnia się wstępnie, czy wykonanie animacji jest możliwe oraz dostarcza nowe klatki w czasie, jaki jest w stanie obsłużyć przeglądarka. Wykorzystałam ją w funkcji *update*, która wywołuje na nowo renderer:

```
1.  update=()=>{
2.      this.renderer.render(this.scene, this.camera);
3.      requestAnimationFrame(this.update);
4.  }
```

Three.js posiada wiele wbudowanych metod i obiektów, które osobie pracującej z 3D w programach graficznych pozwolą intuicyjnie zorientować się w ich wykorzystaniu i funkcjonowaniu. Przykładowo wykonanie oświetlenia w scenie wykonuje się przy pomocy obiektów nazywających się analogicznie do narzędzi software'owych:

```
1.  let hemiLight = new THREE.HemisphereLight(0xffffff,  0.91);
2.  hemiLight.position.set(0, 50, 0);
3.  this.scene.add(hemiLight);
4.  this.dirLight = new THREE.DirectionalLight(0xffffff, 0.84);
5.  this.dirLight.position.set(8, 28, 18);
6.  this.dirLight.castShadow = true;
7.  this.dirLight.shadow.mapSize = new THREE.Vector2(1024, 1024);
```

[24] Dokumentacja MDN, Window.requestAnimationFrame, https://developer.mozilla.org/pl/docs/Web/API/Window/requestAnimationFrame

```
8.  this.scene.add(this.dirLight);
```

1. Tworzę światło sferyczne dookoła sceny
4. Tworzę światło kierunkowe o konkretnych wymiarach

5.3.5 Animowanie modeli 3D w Three.js

W scenie Three.js można korzystać z modeli, a nawet wcześniej przygotowanych animacji tych modeli w programie do grafiki 3D. Opis stworzenia, animowania oraz eksportowania takiego modelu z Maya opisany został powyżej w rozdziale 5.1.5. W tej części opisany jest proces, dzięki któremu zbudowałam drugą scenę projektu, którą jest przedstawienie procentu ubrań używanych przez osobę względem wszystkich posiadanych. Aby to zilustrować, użyłam modelu sukienki, która „wjeżdża" na scenę w pięciu egzemplarzach, z czego po chwili każdy z nich oprócz jednego zmienia kolor aby pokazać, że ubrania które nosimy, stanowią tylko 20% wszystkich, które mamy w szafie. W tym samym momencie animuje się również informacyjny tekst.

Dodanie obiektu do tej sceny wystarczyło wykonać tylko raz. W tym celu użyłam *loader*'a (mechanizmu wgrywania modeli do projektu) do plików w formacie fbx. Wybrałam fbx zamiast gltf gdyż w przypadku mojej animacji okazał się produkować mniejszy objętościowo plik oraz po przetestowaniu sprawiał wrażenie, że tekstura modelu ma lepszy wygląd.

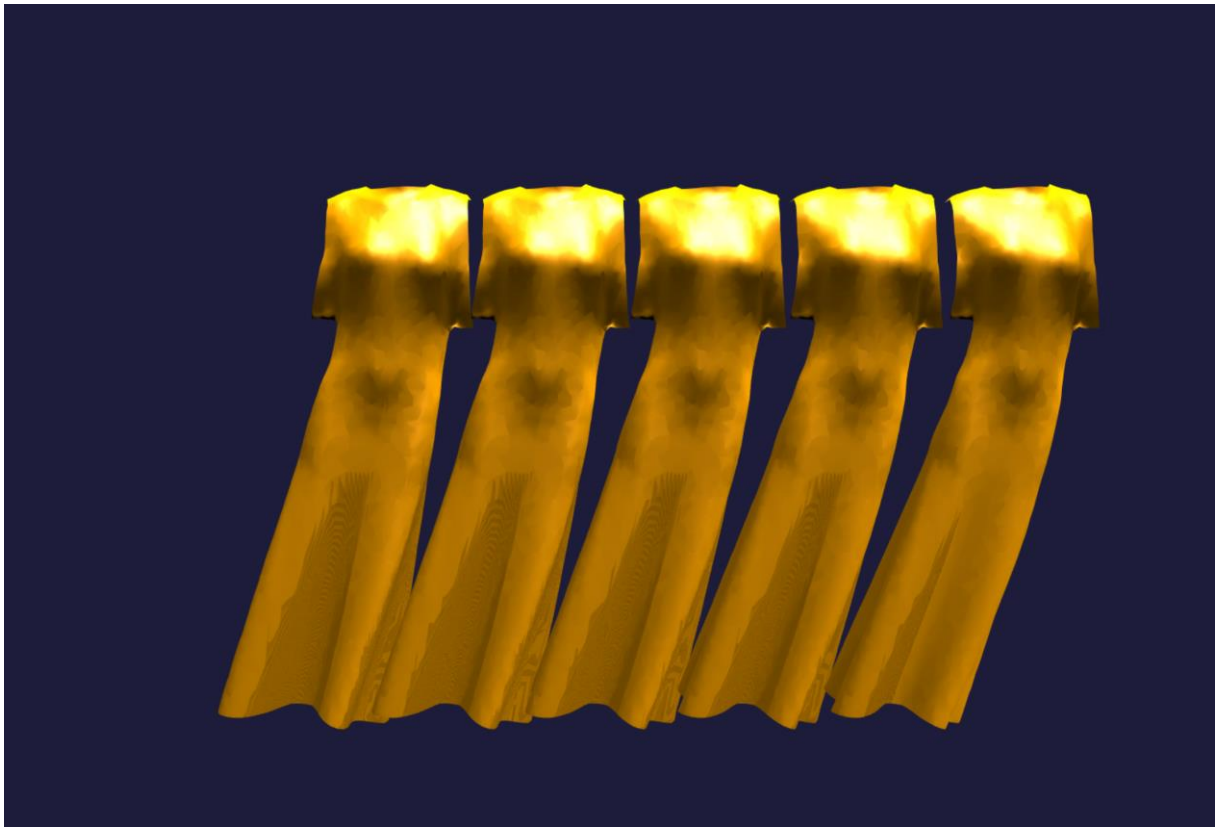
```
1.  this.loader = new THREE.FBXLoader();
2.  this.loader.load(
3.    modelPath,
4.    obj => creationFuntion(obj)
5.    ,undefined,
6.    function(error) {
7.       console.error(error);
8.    }
9.  );
```

1. Tworzę nowy loader, wystarczy jeden na komponent
2. Za pomocą metody ładującej importuję model
3. *modelPath* jest miejscem na ścieżkę do modelu
4. *creationFunction* to funkcja manipulująca modelem po jego załadowaniu, m.in. dodaję w niej model do sceny, zmieniam mu kolor materiału etc.
6. Funkcja obsługująca błąd może równie dobrze zastąpić model inną animacją jeśli byłaby taka potrzeba

W tym miejscu warte podkreślenia jest, że wystarczy tylko raz wgrać model do sceny, by móc wykorzystać go dowolną ilość razy. Czym dokładnie jest wgrywany model .fbx? Po użyciu metody *load*, zwracany jest obiekt 3D, zawierający mnogość atrybutów i informacji, m.in. siatki geometrii, użyte materiały oraz przekształcenia animacji. W celu multiplikacji modelu wystarczy wyekstrahować z obiektu potrzebne dane, a następnie kopiować je w celu powielenia.

```
this.models.push(model);
for(let i =0; i<4; i++){
    let newModel = model.clone();
    newModel.position.x = model.position.x-4*(i+1);
    this.models.push(newModel);
}
```

Przykładowo: zakładając że do *model* jest przypisany argument funkcji *createFunction*, a więc zaimportowanym obiektem fbx, dodaję go do zbioru modeli (tablica *models*). Dodatkowo potrzebuję jeszcze cztery kopie tego modelu, więc w pętli kopiuję go korzystając z metody *clone*. Ponieważ *newModel* jest nową kopią, można zmieniać jego parametry. Przesuwam go w lewo o odległość stosowną do indeksu iteracji. Następnie dodaję do tablicy modeli. Na koniec dodam wszystkie modele z tablicy po kolei również w pętli.



Rysunek 37 Cztery kopie początkowego modelu (ostatni po prawej)

Podobną operację stosuję w przypadku animacji każdego z modeli:

```
                this.models.forEach(model=>{
                   this.scene.add(model);
                   this.mixers.push(new THREE.AnimationMixer(model));
                 })

                 let fileAnimations = obj.animations;
                 let anim = fileAnimations[0];
                 anim.optimize();

                 let modified = {
                   loop : THREE.LoopOnce,
                   clampWhenFinished : true,
                   timeScale : 4
```

```
14.                 }
15.                 this.mixers.forEach(mixer =>{
16.                   this.actions.push(
17.                     overwriteProps(
18.                       mixer.clipAction(anim),
19.                       modified
20.                     )
21.                   )
22.                 })
23.                 this.actions.forEach(action=>{
24.                   action.play();
25.                 });
```

3. Dla każdego z modeli tworzę *AnimationMixer*. Jest to obiekt obsługujący wszystkie animacje podłączone do jednego modelu
6. Wybieram z obiektu fbx animacje dla zaimportowanego modelu
10. Ustawiam parametry animacji – brak pętli, zatrzymanie na ostatniej klatce po ukończeniu oraz spowolnienie animacji
15. Dla każdego z *AnimationMixer*, a więc dla każdego z modeli wyznaczam akcję animacji na podstawie pierwszej animacji przypiętej do obiektu (linijka 7). Funkcja *overwriteProps* dodatkowo nadpisuje właściwości klipu animacji na ustawione w linijce 10
23. Dla każdej z akcji uruchamiam funkcję animacji

Powyższy sposób dokumentuje proces przygotowania i odtworzenia animacji importowanego modelu przy pomocy Three.js. O ile animacja i model są dobrze przygotowane, praca z nimi w kodzie jest bardzo prosta. Możliwości edycyjne dodanego projektu są również dostępne. Przykładowo wzmiankowana wyżej zmiana koloru modeli po zakończeniu modeli uruchomiona jest przez TimelineMax (w zupełnie innym miejscu niż *createFunction*) i wykonuje się za pomocą iteracji po modelach i zmianie właściwości kolorystycznej ich materiału:

```
1.  timeline.to(this.twentyRef, .3, {
2.           opacity: 1,
3.           onStart: ()=>{
4.             this.models.forEach((model, index)=> {
5.
6.               if(index != 2){
7.                 model.traverse(o=>{
8.                   if(o.isMesh){
9.                     o.material = mat2;
10.                  }
11.                })
12.              }
13.            })
14.          }
15.       }, "+=2");
```

Zmienna o nazwie mat2 to zdeklarowany wcześniej materiał typu Phong:

```
1.  const mat2 = new THREE.MeshPhongMaterial(
2.          {
3.              color: 0x1d1c3a,
4.              skinning: true ,
5.              morphTargets :true,
```

```
            specular: 0x009300,
            reflectivity: 0
        });
```

5.3.6 Modyfikacja koloru filmu przy pomocy GLSL

Kolejna scena na stronie wykorzystuje film z eksplozjami sukienek, opisany w rozdziale 5.1.3. Projekt zakłada, że wideo wypełni całą powierzchnię ekranu, a na nim wyświetli się animacja tekstu. Chcąc jednak jak najlepiej wpasować materiał animacji w styl strony chciałam, aby wideo miało przeźroczyste tło. W tym momencie narodził się problem optymalizacyjny – eksport wideo w rozsądnej jakości i wielkości z przeźroczystym tłem generował plik o zbyt dużej wielkości, by wykorzystać go w stronie www. Wyrenderowałam więc animację z domyślnym, czarnym tłem, aby oszczędzić na rozmiarze pliku. W tym rozdziale opiszę, w jaki sposób można użyć shader'a GLSL na ruchomym obrazie tak, aby w wydajny sposób wymienić dowolny kolor na inny – w tym przypadku kolor nieprzeźroczysty (wartość *alpha* równa 1) na kolor przeźroczysty (wartość *alpha* równa 0).

Aby wykorzystać do modyfikacji obrazu shader wideo musi znajdować się w obszarze wpływów WebGL, a więc w kontekście *canvas* HTML5. W związku z tym w tym komponencie również stworzyłam scenę Three.js, do której dodałam płaszczyznę. Kamerę w scenie ustawiłam frontem do płaszczyzny tak, aby ta pokrywała całą powierzchnię *canvas*. Następnie przy użyciu obiektu *VideoTexture* wbudowanego w Three.js utworzyłam nową teksturę z wgranego wcześniej wideo.

W kolejnym kroku stworzyłam nowy materiał obsługujący własne shadery. Do tegoż materiału podpięłam vertex i fragment shader, z czego ten drugi jest wart wytłumaczenia na przykładzie:

```
fragmentShader: 'uniform sampler2D texture;\
            uniform vec3 color;\
            varying vec2 vUv;\
            void main(){\
                vec3 tColor = texture2D( texture, vUv).rgb;\
                float a = (length(tColor - color) - 0.1) * 0.9;\
                gl_FragColor = vec4(tColor, a);}',
```

1. Odwołuję się do zmiennej *texture*, do której w Three.js przypnę teksturę wideo (tj. film)
2. Odwołuję się do zmiennej *color*, w której określę w Three.js kolor do usunięcia z filmu
3. *vUv* to wektor przechowywujący aktualna pozycję na ekranie
5. Przy pomocy funkcji próbkującej zbieram kolor z wideo na pozycji *vUv*
6. Określam wartość zmiennej *a* jeśli odejmę od bieżącego koloru kolor do usunięcia
7. Koloruję obraz w punkcie na obecny kolor tekstury rozszerzony o *a* jako *alpha*, czyli przeźroczystość

Przypisując do materiału zmienne *color* i *texture* uzyskam poprawnie działający shader:

```
let planeMaterial = new THREE.ShaderMaterial({
            uniforms: {
                texture : {
                    type: 't',
                    value: videoTexture
                },
```

```
                color: {
                    type: 'c',
                    value: new THREE.Color(0x000000)
                }
            },  (…)
```

W ostatnim kroku spięłam geometrię płaszczyzny z utworzonym i dodałam ją do sceny, uzyskując wideo z przeźroczystym tłem tam, gdzie wcześniej występowała czerń.

5.3.7 Particle system w Three.js

Podczas opisywania możliwości programu Maya wspomniane było, że particles nie nadają się do eksportu z programu jako animacja. Da się jednak je stworzyć samodzielnie, chociaż nie jest to tak zwięzły proces, jak usuwanie koloru z wideo.

System cząsteczek można stworzyć bez użycia shaderów, jednak jest to całkowicie niewydajne, ponieważ nie otrzymuje wsparcia GPU. Pomoc przy tych shaderach otrzymałam ze strony threejs.org oraz z forum discourse.threejs.org, gdzie twórcy biblioteki pomagają pozostałym adeptom programowania grafiki trójwymiarowej.

Vertex shader:

```
 export default 'attribute float size; \
varying vec3 vColor; \
void main() {\
    vColor = color;\
    vec4 mvPosition = modelViewMatrix * vec4( position, 1.0 );\
    gl_PointSize = size * ( 300.0 / -mvPosition.z );\
    gl_Position = projectionMatrix * mvPosition;\
}';
```

Fragment shader:

```
export default 'uniform sampler2D pointTexture;\
                varying vec3 vColor;\
                void main() {\
                gl_FragColor = vec4( vColor, 0.4 );\
                gl_FragColor = gl_FragColor * texture2D( pointTex-
ture, gl_PointCoord ); \
                }';
```

Shader wierzchołków ustala rozmiar cząstki względem jej oddalenia w osi z (w głąb sceny) na coraz mniejszą względem odległości, ale proporcjonalną do narzuconego z góry *size*, czyli rozmiaru. Fragment shader z kolei pobiera z Three.js teksturę danej cząstki i koloruje nią dany obszar wyznaczony przez *gl_PointCoord*. Predefiniowane zmienne dotyczące punktów mają sens w kontekście obiektów, które z nich korzystają w obrębie Three.js:

```
this.material = new THREE.ShaderMaterial( {
                uniforms: {
                    pointTexture: { value: new THREE.Texture-
Loader().load( "../3d/particle.png" ) }

                },
```

```
                vertexShader: particlesVertexShader,
                fragmentShader: particlesFragmentShader,
                blending: THREE.AdditiveBlending,
                depthTest: false,
                transparent: true,
                vertexColors: true
            });
            this.particleSystem = new THREE.Points(this.geometry, this.material);
```

Na tym przykładzie widać, że Three.js posiada własny, wbudowany generator punktów. Shadery oraz *pointTexture* jedynie nadają mu precyzyjną formę. Oczywiście powyższy kod opisuje wyłącznie materiał particles i potrzebne jest jeszcze stworzenie im geometrii.

Do wykonania geometrii cząsteczek wykorzystałam (wg przykładu wskazanego na stronie three.js.org[25]) *BufferGeometry*. Według opisu z dokumentacji biblioteki, jest to

> (…)wydajna reprezentacja geometrii siatek, linii czy punktów. Zbiera pozycje wierzchołków, wskaźniki ścian, wektorów normalnych, kolorów, UV oraz własnych atrybutów wewnątrz buforów, przez co redukuje koszt przekazywania tych danych do GPU[26].

Dzięki *BufferGeometry* można opisać geometrię cząsteczek, natomiast jest renderowanie przekazać do obsługi shadera, uzyskując najoptymalniejsze efekty. Do osiągnięcia zamierzonego efektu użyłam przykładu Jacka Rugile[27], gdyż pokazywał on w jak sposób uzyskać gładki efekt ruchu cząsteczek względem siebie przy użyciu funkcji kalkulującej przesunięcia *simplex noise*. Od samego stworzenia geometrii, ciekawszy do opisania jest sposób animacji ruchu cząstek przy użyciu właśnie tego *noise*.

Simplex noise to metoda uzyskiwania n-wymiarowego szumu, którą skonstruował Ken Perlin na bazie swojego pierwotnego, powszechnie używanego szumu Perlina. Szum *Simplex* wykorzystuje mniej danych do obliczeń, przez co jest wydajniejszy jeśli nie są potrzebne wyjątkowo precyzyjne rezultaty.

```
let xScaled = part.position.x * noiseScale;
let yScaled = part.position.y * noiseScale;
let zScaled = part.position.z * noiseScale;
let noise1 = this.simplex.noise4D(
                xScaled ,
                yScaled,
                zScaled,
                50 + noiseTime
            )* 0.5;
let noise2 = this.simplex.noise4D(
                xScaled +100,
                yScaled+100,
                zScaled+100,
                50 + noiseTime
            )* 0.5 + 0.5;
let noise3 = this.simplex.noise4D(
                xScaled +200,
                yScaled+200,
                zScaled+200,
                50 + noiseTime
```

[25] Mr Doob, WebGL Custom Attributes Particles, Github 2020, https://github.com/mrdoob/three.js/blob/master/examples/webgl_buffergeometry_custom_attributes_particles.html
[26] Dokumentacja Three.js, BufferGeometry, https://threejs.org/docs/#api/en/core/BufferGeometry
[27] Rugile, Jack, 3D particle explorations, Gtihub 2020, https://github.com/jackrugile/3d-particle-explorations/blob/master/js/demo-8/system.js

```
21.           )* 0.5 + 0.5;
22. part.position.x -= Math.sin(noise1 *Math.PI *2) + noiseVelocity * delta ;
23. part.position.y -= Math.sin(noise2 *Math.PI *2) * noiseVelocity * delta ;
24. part.position.z += Math.sin(noise3 *Math.PI*1.3)  +noiseVelocity * delta ;
```

Płynny ruch cząsteczek w przestrzeni można, wg Rugile, wyliczyć za pomocą szumu jak w przykładzie pokazanym powyżej. Do zmiennych *noise1*, *noise2*, *noise3* liczony jest szum na podstawie wartości pozycji danej cząstki zmniejszonych o pewną skalę. To zmniejszenie ma na celu zniwelowanie gwałtownych przesunięć. Dla każdej ze zmiennych liczony jest szum w czterech wymiarach, tj. trzech na podstawie współrzędnych oraz czwartek z przesunięciem czasu. Warto zaznaczyć, że funkcja zawierająca powyższy kod aktualizowana jest co klatkę animacji, więc wartość *noiseTime* jak i *delta*, które zależne są od upływu czasu, ulegają zmianie. Na podstawie zmiennych szumu w liniach 22-24 ustalane są nowe współrzędne pozycji cząsteczki. Wprowadzona dodatkowo funkcja sinus ma za zadanie zapewnić, że współrzędne będą oscylowały wokół zbliżonych wartości, przez co nie „uciekną” poza obszar sceny.

Jeszcze jedną istotną funkcją wykorzystaną w kodzie Rugile, zmodyfikowaną przeze mnie, jest funkcja aktualizująca geometrię. W powyższym kodzie ustalane były nowe wartości pozycji cząstki, ale należy pamiętać, że cząstka nie jest obiektem samym w sobie, ale składową częścią jednej geometrii, którą trzeba aktualizować po każdej operacji wykonanej na jej elementach.

```
1.  updateParticleAttributes=(color, position, size) =>{
2.   let i = 0;
3.   while(i<this.particles) {
4.      let part = this.parts[i];
5.      if(color) {
6.                const colorAttribute = this.geometry.attributes.color;
7.                      colorAttribute.array[i * 3 + 0] = part.r;
8.                      colorAttribute.array[i * 3 + 1] = part.g;
9.                      colorAttribute.array[i * 3 + 2] = part.b;
10.                     colorAttribute.array[i * 4 + 3] = part.a;
11.               }
12.               if(position) {
13.               const positionAttribute = this.geometry.attributes.position;
14.                     positionAttribute.array[i * 3 + 0] = part.position.x;
15.                     positionAttribute.array[i * 3 + 1] = part.position.y;
16.                     positionAttribute.array[i * 3 + 2] = part.position.z;
17.               }
18.               if(size) {
19.               const sizeAttribute = this.geometry.attributes.size;
20.                     sizeAttribute.array[i] = part.size;
21.           }
22.          i++;
23.    }
24.
25.   if(color) {
26.      this.geometry.attributes.color.needsUpdate = true;
27.   }
28.   if(position) {
29.      this.geometry.attributes.position.needsUpdate = true;
30.   }
31.   if(size) {
32.       this.geometry.attributes.size.needsUpdate = true;
```

```
33.     }
34.       this.geometry.computeBoundingSphere();
35. }
```

Powyższa funkcja jest wywoływana po każdej zmianie na cząstce (będącej w każdym przypadku elementem tablicy *parts*). W liniach 6,13 i 19 pojawia się odwołanie do aktualnego stanu geometrii przez aktualizacją. Następnie w tablicy wektorów przechowującej ten stan aktualizowane są konkretne wartości wg zmian we właściwościach *part[i]*. W kolejnym kroku w liniach 26, 29 i 32 określana jest konieczność aktualizacji kolejnych atrybutów geometrii przy następnym renderze sceny. Linia 34 dodatkowo zawiera wywołanie funkcji przeliczającej całkowity obszar geometrii po tych operacjach.

Przedstawiony kod, będący jedynie fragmentem całego systemu cząstek, prezentuje stosunkowo dużą liczbę iteracji w pętlach na całej liczbie generowanych cząstek (w tym projekcie ta liczba to 10 000). Może to sugerować, że rozwiązanie jest niewydajne, jednak warto zauważyć, co zostało wspomniane już wcześniej – te operacje wykonywane są na buforze danych, więc nie obciążają przeglądarki dodatkową interpretacją wykonanych obliczeń. Za prezentację efektu odpowiada dopiero shader, który na ich podstawie wyświetla obraz przy pomocy GPU, a więc wielowątkowo. Efekt końcowy zatem, chociaż wymaga wielu kroków przygotowania, jest wydajnościowo zadowalający.

5.3.8 Instancje

5.3.8.1 Instancja zorientowana na powierzchnię

W kolejnych sekwencjach wykorzystałam sieci instancji obiektów 3D. Dzięki nim można uzyskać obrazy podobne w strukturze do tych wykonanych z wykorzystaniem MASH, opisanego w rozdziale 5.1.6.1. W tej samej części tekstu przedstawiona jest również logika związana z wykorzystaniem instancji obiektów.

W sekcji mówiącej o produkcji ubrań przez pracownice szwalni ubogich krajów, pozbawione większości praw pracowniczych i egzystujące w naprawdę trudnych warunkach postanowiłam użyć instancji w celu zwielokrotnienia obiektów symbolizujących tysiące uciskanych szwaczek. W tym celu użyłam prostego modelu maszyny do szycia, który powieliłam w sieci dookoła obiektu sukienki jak symbolu marki odzieżowej.

Aby to osiągnąć użyłam przykładu ze strony threejs.org[28], zmodyfikowanego dla uzyskania pożądanego efektu. Do stworzenia geometrii sceny użyłam *InstancedBufferGeometry*, która dziedziczy bezpośrednio po *BufferGeometry* opisanej w rozdziale wcześniej, przy czym obsługuje tworzenie instancji na bazie istniejącego już modelu. Model do stworzenia instancji zaimportowałam przez *FBXLoader*. Następnie, wg przykładu z threejs.org, skopiowałam do prototypu obiektu modelu cechy *InstancedBufferGeometry*. W kolejnym kroku, po transformacji wg uznania, utworzyłam *InstancedMesh* z tejże geometrii oraz z materiału Lambert z randomizowaną wartością koloru:

[28] Mr Doob, WebGL Instancing Scatter, Github 2020, https://github.com/mrdoob/three.js/blob/master/examples/webgl_instancing_scatter.html

```
1.  this.modelGeometry = new THREE.InstancedBufferGeometry();
2.  THREE.BufferGeometry.prototype.copy.call(  this.modelGeometry , this.machine.chil-
    dren[4].geometry );
3.  var defaultTransform = new THREE.Matrix4()
4.                  .makeRotationX( Math.PI )
5.                  .multiply( new THREE.Matrix4().makeScale( 1, 1, 1 ) )
6.                  .makeTranslation(81,-1000,0);
7.  this.modelGeometry.applyMatrix4(defaultTransform);
8.  (...)
9.  this.modelMesh = new THREE.InstancedMesh(this.modelGeometry, this.model-
    Material, this.state.count);
```

Po stworzeniu sieci instancji istotne było sprawienie, by sieć ta adaptowała się do sieci modelu sukienki. Do uzyskania zamierzonego efektu ważne było, aby maszyny skupiały się wokół tego modelu. W tym celu autorzy przykładu wykorzystali *MeshSurfaceSampler*, czyli mechanizm próbkowania powierzchni sieci geometrycznej, którą można skopiować na inny obiekt. W tym celu w kodzie strony wykorzystuję funkcję *resampleParticle*, która dla poszczególnej instancji wykonuje próbkowanie sieci powierzchni i kopiuje na instancję właściwości pozycji i wektora normalnego tak, aby ta była przyległa do powierzchni lub proporcjonalnie od niej oddalona i ukierunkowana. Próbkowanie odbywa się przy pomocy funkcji *resample,* która poza sprawdzeniem właściwości sieci powierzchni nadaje instancjom zróżnicowaną skalę i czas istnienia:

```
1.  resample = () => {
2.        this.sampler = new MeshSurfaceSampler(this.surface.children[0]).build();
3.
4.        for( let i =0; i< this.state.count; i++){
5.
6.            this.ages[i] = Math.random();
7.            this.scales[i] = scaleCurve(this.ages[i]);
8.            this.resampleParticle(i);
9.
10.       }
11.       this.modelMesh.instanceMatrix.needsUpdate = true;
12.       this.scene.add(this.modelMesh);
13.   }
14.   resampleParticle = i => {
15.       this.sampler.sample(this._position, this._normal);
16.       this._normal.add(this._position);
17.       this.dummy.position.copy(this._position);
18.
19.       this.dummy.scale.set( this.scales[ i ], this.scales[ i ], this.scales[ i ] );

20.       this.dummy.lookAt( this._normal );
21.       this.dummy.updateMatrix();
22.       this.modelMesh.setMatrixAt(i, this.dummy.matrix);
23.
24.   }
```

W linii 11 widzimy odwołanie do matrycy instancji w celu uruchomienia jej aktualizacji. Ta sama metoda wykorzystana jest wszędzie indziej, gdzie modyfikuję wartości instancji, podobnie jak w przypadku particles.

W metodzie rysującej scenę wykorzystana jest dodatkowo funkcja *updateParticle*, która właśnie modyfikuje właściwości konkretnej instancji, a potem również odwołuje się do *resampleParticle* aby poprawnie umiejscowić instancję w scenie:

```
1.  updateParticle = i => {
2.          this.ages[i] += 0.00005;
3.          if(this.ages[i] >= 1){
4.
5.              this.ages[i] = 0.001;
6.             this.scales[i] = scaleCurve( this.ages[i]);
7.
8.                  this.resampleParticle(i);
9.              return;
10.         }
11.         let prevScale = this.scales[i];
12.         this.scales[i] = scaleCurve(this.ages[i]);
13.         this._scale.set(this.scales[i]/prevScale,this.scales[i]/prevScale,this.scale
    s[i]/prevScale);
14.
15.         this.modelMesh.getMatrixAt(i, this.dummy.matrix);
16.         this.dummy.matrix.scale(this._scale);
17.         this.modelMesh.setMatrixAt(i, this.dummy.matrix);
18.     }
```

1. W liniach 1-5 manipuluję wartością *ages[i]* by zachować fluktuację instancji
6. Skala instancji ustalana jest na podstawie jej wieku. W liniach 11-13 również
8. Wywołuję metodę *resampleParticle* aby sprawdzić miejsce dla instancji

W ten sposób został uzyskany efekt powielenia instancji maszyny do 100 000 sztuk z zachowaniem wydajnego działania strony w przeglądarce:

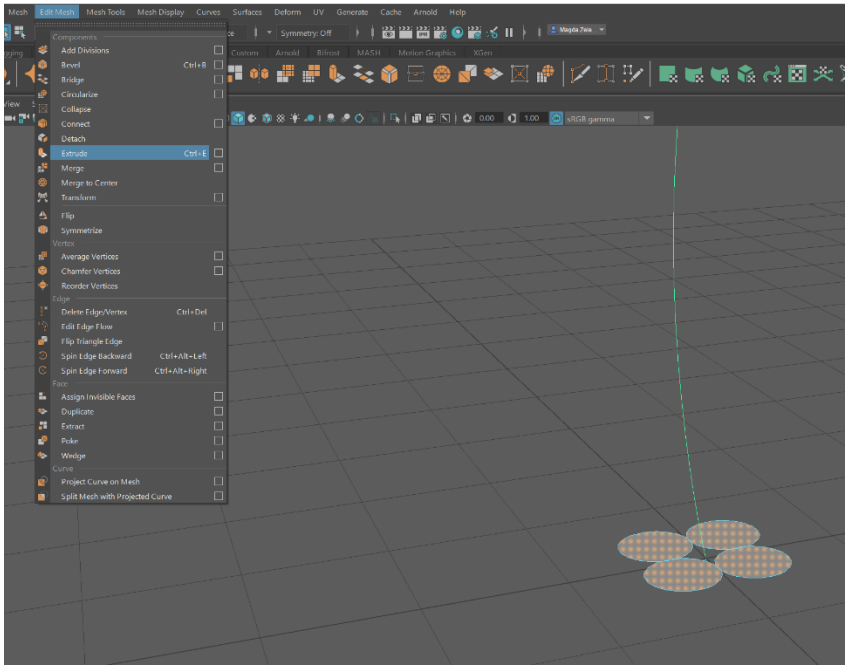
Rysunek 38 Scena z wykorzystaniem instancji zorientowanych na powierzchnię

5.3.8.2 Instancje z kontrolowanym umieszczeniem

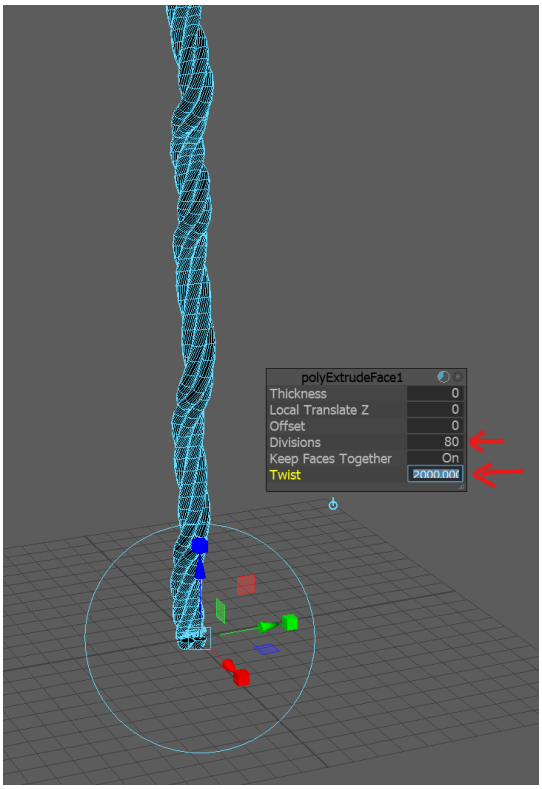
Przykład powyżej realizował skomplikowany układ instancji zorientowanych na konkretny obiekt w przestrzeni. Prostszym, ale równie ciekawym przykładem zastosowania instancji jest wykorzystanie ich do stworzenia zupełnie nowych obiektów.

Kolejna scena projektu opisuje mierną jakość tekstyliów wykorzystywanych w branży modowej, co sprawia, że ubrania są nietrwałe i szybko nadają się do wyrzucenia. Do zilustrowania tej sytuacji wybrałam zbliżenie na siatkę włókien tkaniny, które będą przesuwały się względem siebie, sugerując nietrwałość. Model siatki z satysfakcjonującą liczbą detalu oraz z ciekawym przesunięciem wykonałam najpierw przy użyciu MASH, ale jego eksport wygenerował by zbyt duży plik, by można z niego wygodnie skorzystać w stronie www.

Do powielenia tego efektu wykorzystałam więc instancje. Eksportowałam z Maya pojedynczą nić siatki. Wykonałam ją wytłaczając cylindry na bazie czterech owali wzdłuż krzywej, które to wytłoczenie następnie skręciłam wokół osi krzywej.



Rysunek 39 Wytłaczanie cylindrów z wykorzystaniem opcji Extrude



Rysunek 40 Skręt cylindrów wokół krzywej

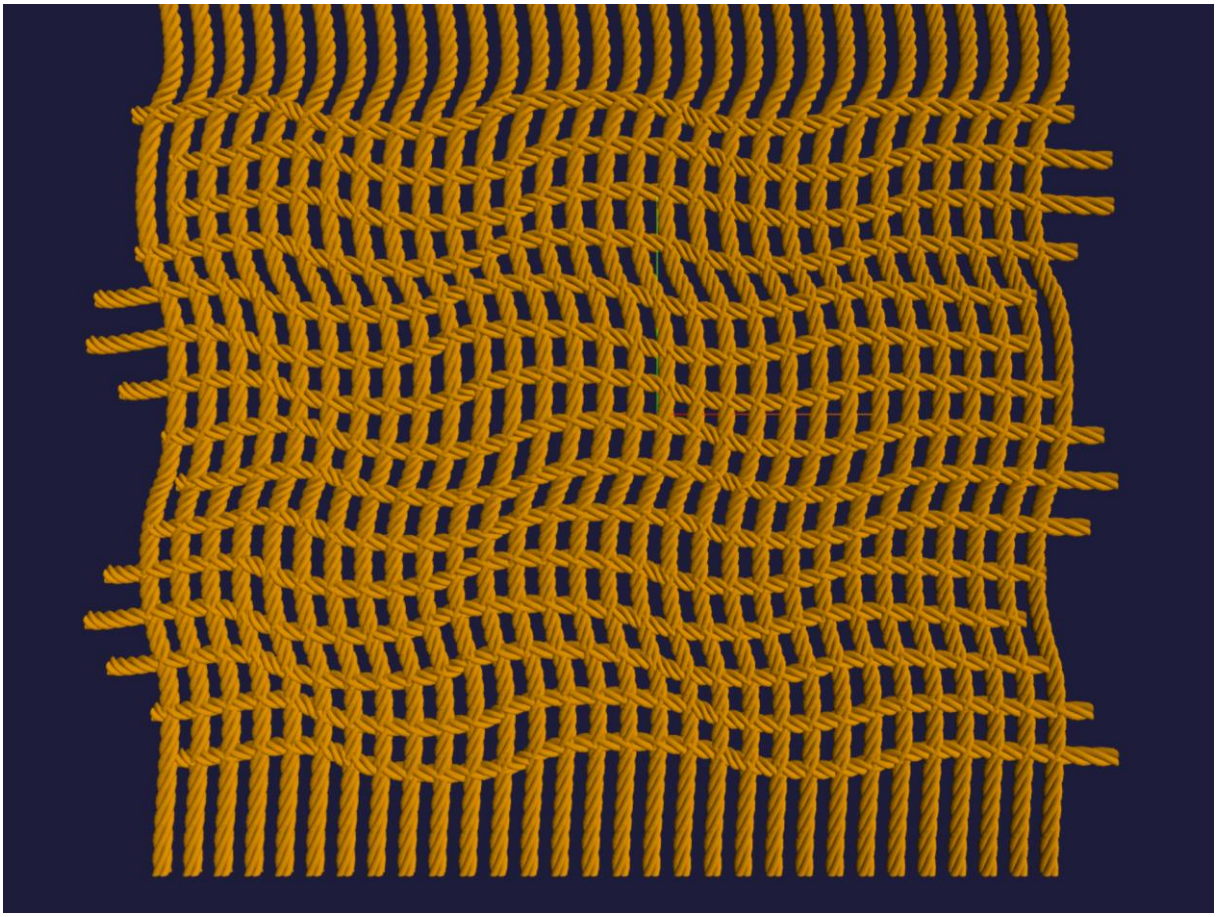
Po uzyskaniu modelu „nici", wyeksportowałam go z Maya w formacie fbx, by następnie załadować go do sceny Three.js według wielokrotnie opisywanego już schematu.

Na bazie tej gotowej geometrii stworzyłam *InstancedMesh*. Użycie buforów było na pewno wydajniejsze, ale w tym przypadku korzystam tylko z kilkunastu instancji, więc uznałam różnicę za pomijalną.

Proces ustawienia instancji względem siebie wykonałam już w funkcji rysującej, gdyż elementy mają być przesuwane w czasie. Korzystając z pętli po ilości pożądanych nitek w imitacji materiału, rozstawiłam instancje względem siebie z drobnym przesunięciem. Połowę z nich, korzystając z warunku na bazie modulo, obróciłam o 90 stopni w osi z, przez co zbiory nici znalazły się do siebie prostopadle:

```
            var offset = ( this.amount - 1 ) / 20;
            for ( var x = 0; x < this.amount; x ++ ) {
                for ( var y = 0; y < this.amount; y ++ ) {
                    if(i % 2 === 0){
                        this.dummy.rotation.z = Math.PI/2;
                        this.dummy.position.set(2+ Math.sin(y/2+ delta ) , offset-
    y/2, 0);
                    }else{
                        this.dummy.rotation.z = 0;
                        this.dummy.position.set(offset - x/1.5, -15, 0);
                    }
                        this.dummy.updateMatrix();
                        this.mesh.setMatrixAt( i ++, this.dummy.matrix );
                }
            }
            this.mesh.instanceMatrix.needsUpdate = true;
        }
```

Efekt końcowy okazał się identyczny jak w przypadku prób wykonywanych w Maya MASH:



Rysunek 41 Sieć instancji

5.3.8.3 Particle system przy użyciu instancji

Ostatnia scena wykorzystuje dla porównania z particle system utworzonym przy pomocy Points i shaderów wykorzystuje w tym celu instancje. Przykład ten również jest zaczerpnięty z threejs.org[29] i jest bardzo ciekawy do opisania.

Konstrukcja tego przykładu jest o wiele prostsza niż w przypadku systemu cząsteczek opisanego wcześniej. Ten kod również wykorzystuje geometrię w buforach *BufferGeometry*, ale zaczyna się od stworzenia bardzo zrandomizowanego zbioru punktów w przestrzeni:

```
1.  let vector = new THREE.Vector4();
2.          const instances = 500;
3.          let positions = [];
4.          let offsets = [];
5.          let colors = [];
6.          let orientationsStart = [];
7.          let orientationsEnd = [];
```

[29] Mr Doob, BufferGeometry/Instancing, Threejs 2020, https://threejs.org/examples/?q=ins#webgl_buffergeometry_instancing

```
8.
9.          positions.push( 0.025, - 0.025, 0 );
10.         positions.push( - 0.025, 0.025, 0 );
11.         positions.push( 0, 0, 0.025 );
12.
13.
14. for(let i = 0; i< instances; i++){
15.    offsets.push(Math.random() - 0.5, Math.random() - 0.5, Math.random() - 0.5);
16.    colors.push(Math.random()/5, Math.random()/3, Math.random(), Math.random());
17.    vector.set( Math.random() * 2 - 1, Math.random() * 2 - 1, Math.random() * 2 - 1,
    Math.random() * 2 - 1 );
18.    vector.normalize();
19.    orientationsStart.push(vector.x, vector.y, vector.z, vector.w);
20.    vector.set( Math.random() * 2 - 1, Math.random() * 2 - 1, Math.ran-
    dom() * 2 - 1, Math.random() * 2 - 1 );
21.    vector.normalize();
22.    orientationsEnd.push(vector.x, vector.y, vector.z, vector.w);
23. }
24. const geometry = new THREE.InstancedBufferGeometry();
25. geometry.instanceCount = instances;
26. geometry.setAttribute('position', new THREE.Float32BufferAttribute(positions, 3));
27. geometry.setAttribute( 'offset', new THREE.InstancedBufferAttribute( new Float32Ar-
    ray( offsets ), 3 ) );
28. geometry.setAttribute( 'color', new THREE.InstancedBufferAttribute( new Float32Ar-
    ray( colors ), 4 ) );
29. geometry.setAttribute( 'orientationStart', new THREE.InstancedBufferAttrib-
    ute( new Float32Array( orientationsStart ), 4 ) );
30. geometry.setAttribute( 'orientationEnd', new THREE.InstancedBufferAttrib-
    ute( new Float32Array( orientationsEnd ), 4 ) );
```

Posiadając taką geometrię, do uzyskania efektów wizualnych można użyć materiału z własnymi shaderami. Shadery zaproponowane w tym przykładzie przez twórców three.js są o wiele ciekawsze niż wykorzystywane wcześniej gdyż po pierwsze odwołują się bezpośrednio do geometrii, a po drugie ich zmienne są animowane w czasie przez funkcję rysującą. Opis warto zacząć od shadera wierzchołków:

```
1.  export default 'precision highp float;\
2.  \
3.  uniform float sineTime;\
4.  \
5.  uniform mat4 modelViewMatrix;\
6.  uniform mat4 projectionMatrix;\
7.  \
8.  attribute vec3 position;\
9.  attribute vec3 offset;\
10. attribute vec4 color;\
11. attribute vec4 orientationStart;\
12. attribute vec4 orientationEnd;\
13. \
14. varying vec3 vPosition;\
15. varying vec4 vColor;\
16. \
17. void main(){\
18. \
19.     vPosition = offset * max( abs( sineTime * 2.0 + 1.0 ), 0.5 ) + position;\
20.     vec4 orientation = normalize( mix( orientationStart, orientationEnd, sine-
    Time ) );\
21.     vec3 vcV = cross( orientation.xyz, vPosition );\
22.     vPosition = vcV * ( 2.0 * orientation.w ) + ( cross( orienta-
    tion.xyz, vcV ) * 2.0 + vPosition );\
23. \
```

```
24.     vColor = color;\
25. \
26.     gl_Position = projectionMatrix * modelViewMatrix * vec4( vPosition, 1.0 );\
27. \
28. }'
```

Linia 3 zawiera odwołanie do zmiennej *sineTime*, która przeliczana będzie w funkcji rysującej. Linie 8-12 stanowią odwołania do atrybutów geometrii, jak pokazano na przykładzie powyżej. Funkcja *main* ma za zadanie wyliczyć nową pozycję wierzchołka względem poprzedniej oraz zmiennej *sineTime*. Dodatkowo, za pomocą zmiennych *orientationStart* i *orientationEnd* policzyć zwrot ruchu wierzchołka i ewentualnie go zmienić.

Fragment shader:

```
1.  export default 'precision highp float;\
2.  \
3.  uniform float time;\
4.  \
5.  varying vec3 vPosition;\
6.  varying vec4 vColor;\
7.  \
8.  void main() {\
9.  \
10.     vec4 color = vec4( vColor );\
11.     color.b += sin( vPosition.x * 180.0 + time ) * 0.5;\
12.     color.r -= sin( vPosition.y * 1.0 +time ) * 0.1;\
13.     color.g -= sin( vPosition.z * 2.0 +time) * 0.1;\
14.     gl_FragColor = color;\
15. \
16. }';
```

Fragment shader natomiast wykorzystuje zmienną *time* określającą upływ czasu. Względem jej wartości oraz aktualnej pozycji wierzchołka wyznacza zmianę koloru wierzchołka, przez to system cząstek może mienić się kolorami w częstotliwości zależnej od odległości.

Obie wspomniane zmienne *time* oraz *sineTime* są przeliczane w funkcji rysującej *update*:

```
1.  let delta = performance.now();
2.  let obj = this.scene.children[1];
3.  obj.rotation.y = delta * 0.0005;
4.  obj.material.uniforms["time"].value = delta * 0.005;
5.  obj.material.uniforms["sineTime"].value = Math.sin( obj.material.uni-
    forms[ "time" ].value * 0.05 );
```

W ten sposób geometria oraz materiał siatki są animowane w czasie tworząc wydajne efekty przy pomocy shaderów. Przygotowanie i obsługa zmian geometrii jest tutaj wydajniejsza niż w przypadku particle system stworzonego za pomocą Points, ponieważ zmiany w pozycji obsługiwane są przez shader. Ta technika wymaga jednak bardziej zaawansowanej znajomości korzystania z shaderów oraz języka GLSL.

5.3.9 PIXI.js i image displacement

Na końcu witryny znajduje się dodatkowa sekcja zawierająca bibliografię i bazę wiedzy na temat przemysłu modowego i jego eksploatacji środowiska, a także przydatne informacje jak zachowywać się w obliczu tego problemu.

Wśród materiałów rozróżniłam na kilka kategorii tematycznych oraz dwa typu zawartości: wideo oraz dokument. Zebrałam wszystkie linki oraz informacje w jeden plik json, z którego wyświetlałam te dane na widoku. Tytuły kategorii umieściłam w słowniku.

Z tablicy materiałów wygenerowałam trzy poziome kontenery (bazując na kategoriach), każdy z nich o identycznym wyglądzie, które prezentowały obraz wyróżniający dany link obramowany hiperłączem oraz teksty informacyjne pod spodem. Materiały wyświetlały się do siebie równolegle.

Przewidziałam, że materiałów w kategorii może być więcej niż zmieści się w poziomie na ekranie. Do każdego z kontenerów dodałam slider reagujący na przeciąganie elementu przytrzymując klawisz myszy (lub palec na urządzeniach dotykowych). Na slider składają się trzy funkcje: *startDrag, stopDrag* oraz *dragDiv*. Pierwsza funkcja przypięta jest do zdarzenia wciśnięcia przycisku myszy. Po wywołaniu sprawdza, czy element na którym naciśnieto jest przesuwny, a jeśli tak, zapamiętywane są jego współrzędne przesunięcia z lewej strony (inicjalnie jest to 0) i wywoływana jest funkcja *dragDiv* przypięta do zdarzenia ruchu kursorem. Ta funkcja na bieżąco zmienia pozycję kontenera zgodnie ze zmianą pozycji kursora oraz zapamiętuje finalną wartość przesunięcia, a także kierunek. Po uwolnieniu przycisku myszy uruchamiana jest spięta z tym zdarzeniem funkcja *stopDrag,* która kończy nasłuchiwanie zmian pozycji kursora.

Do przesuwanych obrazów chciałam dodać efekt falowania zbliżony do ruchu materiału wjeżdżających sukienek w sekcji drugiej. Aby to osiągnąć, skorzystałam z biblioteki Pixi.js, która podobnie jak Three.js, korzysta ze standardu WebGL, ale nie wymaga konstruowania całej sceny 3D do osiągnięcia prostego efektu.

Do pracy z Pixi, zgodnie z definicją WebGL, potrzebny jest kontekst *Canvas* HTML5. Ponieważ API Pixi zakłada z góry, że jeśli będziemy coś w nim tworzyć, potrzebujemy *Canvas,* funkcja uruchamiająca renderer automatycznie ten element tworzy. Aby animował się każdy obrazek w sliderze, na każdym obiekcie z materiałami dodatkowymi należy taki obiekt utworzyć. Całą operację inicjalizacji i ustawień wykonałam iterując po tablicy z elementami slidera.

Do osiągnięcia efektu falowania wykorzystałam technikę nazwaną *displacement* (przemieszczenie). Polega ona na wykorzystaniu dwóch tekstur: tekstury zdjęcia czy obrazu, który chcemy „animować" – tekstury A - oraz tekstury wzoru czy szumu, preferencyjnie czarno-białej, na bazie której utworzone zostaną specjalne efekty – tekstury B. Ta tekstura jest niewidoczna, ale wykorzystywana przez shader to przemieszczania poszczególnych fragmentów tekstury A. Odbywa się to w ten sposób, że czytana jest wartość jasności piksela tekstury B, po czym bezpośrednio mapując pozycję tego piksela na teksturę A, przesuwana jest pozycja korespondującego piksela A proporcjonalnie do wartości

jasności. Jeśli dodatkowo nadamy teksturze B jakiś ruch, przykładowo przesunięcie lub obrót, uzyskamy bardziej lub mniej wyraziste efekty na teksturze A.

```
const thumbs = findRef.current.getElementsByClassName('thumb');

Array.from(thumbs).map((item, index) => {
            let playground = {};
            playground.renderer = PIXI.autoDetectRenderer({width: thumbs[index].offsetWidth, height: thumbs[index].offsetHeight, transparent:true});
            playground.renderer.autoResize = true;
            item.appendChild(playground.renderer.view);
            let tp = PIXI.Texture.from(item.dataset.path);

            let preview = new PIXI.Sprite(tp);
            preview.anchor.set(0.5);
            preview.width = playground.renderer.width;
            preview.height = playground.renderer.height;
            preview.x = playground.renderer.width / 2 ;
            preview.y = playground.renderer.height /2;
            playground.preview = preview;

            let displacementSprite = PIXI.Sprite.from('./images/wrinkles.jpg');
            displacementSprite.texture.baseTexture.wrapMode = PIXI.WRAP_MODES.REPEAT;
            let displacementFilter = new PIXI.filters.DisplacementFilter(displacementSprite);
            displacementSprite.scale.y = 0;
            displacementSprite.scale.x = 0;
            displacementSprite.rotation = 90;
            let stage = new PIXI.Container();
            stage.addChild(displacementSprite);

            stage.addChild(preview);
            playground.displacementSprite = displacementSprite;
            displacementFilter.autoFit = false;
            displacementFilter.padding = 50;
            playground.displacementFilter = displacementFilter;
            playground.stage = stage;
            playground.category = item.dataset.category;
            playgrounds.push(playground);
            animate();
        })
```

Zależność intensywności animacji względem intensywności ruchu tekstury B wykorzystałam w tym przykładzie. Inicjalnie, skala filtra *displacement* jest równa 0 i jest on niewidoczny. Dopiero podczas ruchu przesuwania animowane są jego wartości. Uzależniłam je dodatkowo od kierunku oraz dystansu przesunięcia, aby dostarczyć jak najlepszej dynamiki do efektu:

```
TweenMax.to(item.renderer.view, .5,
  {skewX: direction}); //przechyl w kierunku ruchu
TweenMax.to(item.displacementSprite, 3,
  {rotation: 90+10*direction/distance*Math.random()});//obróć teksturę w kierunku
TweenMax.to(item.displacementSprite.scale,2,
  {x:Math.random()+direction/distance*Math.random(),
   y:1+direction/distance*Math.random(), //wartości muszą oscylować między 0 a 2 aby efekt był realistyczny
  ease: ease});
```

Dodatkowo dodałam jeszcze intensywniejsze poruszenie tekstury na *startDrag* oraz szybkie wyciszenie ruchu na *stopDrag*. Efekt powstał jak następuje:

Rysunek 42 DisplacementFilter na sliderze

5.4 PODSUMOWANIE

Praca z trójwymiarowością w przeglądarce nie należy być może do prostych i z całą pewnością należy poświęcić wiele linii kodu do uzyskania wizualnie niewielkich efektów. Jednak równocześnie daje możliwości odtwarzania w czasie rzeczywistym animacji, które do tej pory były poza zasięgiem przeglądarek www. Strona informacyjna jak w projekcie nie jest najwyższej próby wyzwaniem dla technologii WebGL, gdyż tworzone są pełne gry 3D lub widowiska VR, jednak za jej pomocą udało mi się przyjrzeć z bliska wielu technikom, z których do tej pory korzystałam przez interfejs programów graficznych.

6 WNIOSKI KOŃCOWE

Praca z grafiką 3D jest zdecydowanie różna od pracy z grafiką 2D i bynajmniej nie jest to kwestia rozszerzenia przestrzeni twórczej o jeden wymiar. Grafika trójwymiarowa wymaga przynajmniej podstawowej znajomości praw fizyki, konstrukcji komputera, matematyki oraz nieraz skomplikowanych konceptów dotyczących niej samej.

Niezaprzeczalne jednak jest, że oferuje potężne możliwości twórcom wizualnym, jak i programistom. W tym projekcie stworzyłam mechanizmy, które de facto przypominają interaktywny film animowany bardziej niż klasyczną witrynę internetową. Dodatkowo, w kwestii optymalizacyjnej, strona rażąco nie wykracza poza standardy mocy obliczeniowej przewidziane dla jej typu.

Uważam, że nauka software'u do pracy z grafiką 3D jest żmudna i trudna, ale możliwa do opanowania. Wymaga cierpliwości, systematyczności, ale przede wszystkim mentoringu osoby zaznajomionej profesjonalnie z jej pryncypiami. Na bazie samej dokumentacji ciężko jest posiąść stosowną wiedzę, zwłaszcza w zakresie optymalizacji pracy. Ja miałam komfort przyswajania tej wiedzy w środowisku akademickim.

Nauka programowania z użyciem grafiki 3D prezentuje się podobnie. Bez wsparcia twórców Three.js na forum biblioteki oraz przykładów kodu, które umieścili na GitHub, nauka z samej dokumentacji byłaby odkrywaniem nowego lądu. Niewątpliwie ułatwiające jest posiadanie wiedzy z obsługi programów, ale niewystarczające.

Podczas tworzenia tej pracy opanowałam wiele technik, o których uczono lub tylko wspominano w trakcie studiów lub w ogóle nie wchodziły w zakres curriculum. Przetworzyłam i zaadaptowałam liczbę przykładów zaczerpniętych od najlepszych w dziedzinie po to, by uzyskać interesującą wizualnie formę informacyjną. Myślę, że jeszcze wiele elementów kodu wymaga poprawy czy optymalizacji, ale wynika to z tego, że apetyt na wiedzę rośnie w miarę jedzenia, a informacje, które posiadam teraz, nie były dla mnie dostępne gdy zaczynałam.

Gdybym rozpoczęła pracę nad projektem jeszcze raz, jednak z wiedzą, którą już posiadam, w kilku miejscach przygotowałabym się inaczej:

1. **Przygotowanie modeli i animacji 3D do użycia w kodzie WebGL** – animacje modeli w tym projekcie były przygotowywane przeze mnie ze zbyt dużą dbałością o szczegóły i płynność, by miało to uzasadnienie. Niestety obecne możliwości sprzętowe dotyczące przeglądarek www nie są w stanie obsłużyć płynnych animacji 3D tak, jak umożliwia to dedykowany software. Uważam, że w pracy zbyt dużo czasu poświęciłam na dopieszczanie modeli, które w zamyśle projektowym miały nie być użyte jako wideo. Czas ten poświęcić bym mogła na manipulację kodem Three.js od uzyskania efektów nie obciążających zbyt CPU.
2. **Projekt UX strony** – założenie co do skupienia się na obsłudze strony głównie poprzez przewijanie było błędne, a próby załatania dziur wynikających z braku obsługi różnych

metod nawigacji przez użytkownika dały dość chaotyczne efekty. Strona jest na tyle skomplikowana, a także podzielona na konkretne, kompletne sekcje, że płynne przejścia pomiędzy nimi są niemalże niemożliwe do uzyskania przy zachowaniu pełnej dostępności.

3. **Przygotowanie animacji 3D w formie wideo** – przygotowując animacje skupiłam się głównie na jakości i detalu modeli, pomijając aspekt oświetlenia i kontrastu. W celu użycia wideo na stronie musiałam znacznie obniżyć jego jakość by zoptymalizować wielkość pliku, przez co dopracowane szczegóły są niewidoczne. Uważam, że w takim użyciu wideo powinno być przygotowane z pewnym przerysowaniem kontrastu, a jednak uproszczeniem formy, tak, aby efektowniej prezentowało się przy spadku jakości.
4. **Projekt przejść między sekcjami** – przygotowując projekt założyłam, że wszystkie sekcje będą dostępne w treści dokumentu strony i będę w stanie płynnie animować przejścia między nimi. Założenie to pominęło możliwości sprzętowe przeglądarek i o ile w wersji deweloperskiej prezentowało się dostatecznie dobrze, po umieszczeniu na serwerze okazało się, że renderowanie wszystkich sekcji na raz jest zabójcze dla wydajności sprzętowej. W efekcie każda sekcja musi być renderowana oddzielnie, zależnie od widoku, przez co nie mam możliwości uzyskania efektownych przejść między nimi. Zaczynając pracę na nowo zredukowałabym ilość różnych technik i współdzieliła kod na wszystkie sekcje.
5. **Użycie programu Maya** – do przygotowania modeli do użycia przez WebGL następnym razem wykorzystałabym program Blender, a nie Maya. Maya jest świetnym oprogramowaniem, jednak ukierunkowanym na zupełnie inny rodzaj pracy, przy czym Blender ma pełne wsparcie przez twórców Three.js i jest mocno ukierunkowany na pracę w sieci.

6.1 CZEGO NIE UDAŁO SIĘ OSIĄGNĄĆ

Chwalę sobie możliwość wykorzystania i zaprezentowania w tym projekcie wielu technik z wykorzystaniem WebGL, jednakże uważam, że forma przesłoniła tutaj treść. Wprowadzając wiele rozwiązań wizualnych uniemożliwiłam wprowadzenie większej ilości tekstu, przez co nie byłam w stanie przekazać tylu informacji ile bym chciała. Dodatkowo napotkałam dużą trudność w podaniu treści tekstowych tak, by przykuwały uwagę użytkownika, nie ginąc w obliczu efektów generowanych przez grafikę 3D. Wydaje mi się, że oddala to projekt od pierwotnego założenia, jakim miała być prezentacja informacji przy wsparciu WebGL.

Przy projektowaniu strony zakładałam też większy udział płynnych animacji modeli, co okazało się być bardzo trudne do zrealizowania z przyczyn technicznych. Napotkałam kłopoty w trakcie przygotowania i eksportu animacji przez program Maya. Tak jak pisałam powyżej, nie użyłabym Maya do takiej pracy ponownie. Przez napotkane kłopoty musiałam pozbyć się kilku efektów przejść, np. w animacji wjazdu sukienek na scenę, gdyż walka z oprogramowaniem zajmowała zbyt dużo czasu, a w efekcie źródło problemu zidentyfikowałam jako wada wtyczki do eksportu FBX.

6.2 DALSZY ROZWÓJ PROJEKTU

Projekt jest obszernym przeglądem różnych technik 3D i zasługuje na publikację w ramach prezentacji tych możliwości. Aby jednak był wygodny w użyciu oraz spełniał pierwotne założenia projektowe, wymagany jest szereg modyfikacji:

1. Poprawa UX – obsługa przejść poprzez przewijanie jest niewystarczająca. Zamierzam dodać dodatkowe możliwości nawigacji w projekcie, aby zapewnić lepszą kontrolę użytkowania.
2. Dodanie menu – zamierzam rozszerzyć nawigację o menu, w którym poza ułatwionym dostępem do sekcji pojawią się linki do podstron zawierających między innymi źródła prezentowanych treści.
3. Responsywność – w obecnym kształcie strona nie jest w pełni responsywna, tj. nie wyświetla się poprawnie na małych ekranach urządzeń przenośnych. Dodatkowo nie posiada wsparcia dla starszych przeglądarek oraz tych, mających kłopoty w WebGL. Zamierzam poświęcić się przygotowaniu jej do pełni możliwości webowych oraz wykonać szereg testów między przeglądarkami.
4. Rozszerzenie treści – ponieważ, jak zostało to opisane powyżej, należało ograniczyć prezentowane treści by nie wprowadzać chaosu, zamierzam rozszerzyć udostępniane przez stronę informacje na zasadzie tekstowych podstron, do których dostęp będzie przy ich syntetycznych streszczeniach na stronie głównej.
5. Optymalizacja – zamierzam wprowadzić dodatkowe zabiegi optymalizacyjne, takie jak współdzielenie zasobów przez sekcje, progresywne ładowanie zasobów, redukcja obliczeń.

Liczę, że dzięki powyższym zabiegom projekt stanie się w rzeczywistości użyteczną stroną www z wykorzystaniem efektownych zabiegów graficznych.

7 SPIS ILUSTRACJI

8 BIBLIOGRAFIA

1. Parisi, Tony, *Aplikacje 3D. Przewodnik po HTML5, WebGL i CSS3*, Wydawnictwo HELION, Warszawa 2015
2. Mr Doob, *Three.js docs*, www.threejs.org/docs (dostęp 02.06.2020)
3. *Mozilla Developer Network*, www.mdn.com (dostęp 02.06.2020)