Hak cipta dan penggunaan kembali:

Lisensi ini mengizinkan setiap orang untuk menggubah, memperbaiki, dan membuat ciptaan turunan bukan untuk kepentingan komersial, selama anda mencantumkan nama penulis dan melisensikan ciptaan turunan dengan syarat yang serupa dengan ciptaan asli.

Copyright and reuse:

This license lets you remix, tweak, and build upon work non-commercially, as long as you credit the origin creator and license it on your new creations under the identical terms.

Perpustakaan Univeristas Multimedia Nusantara

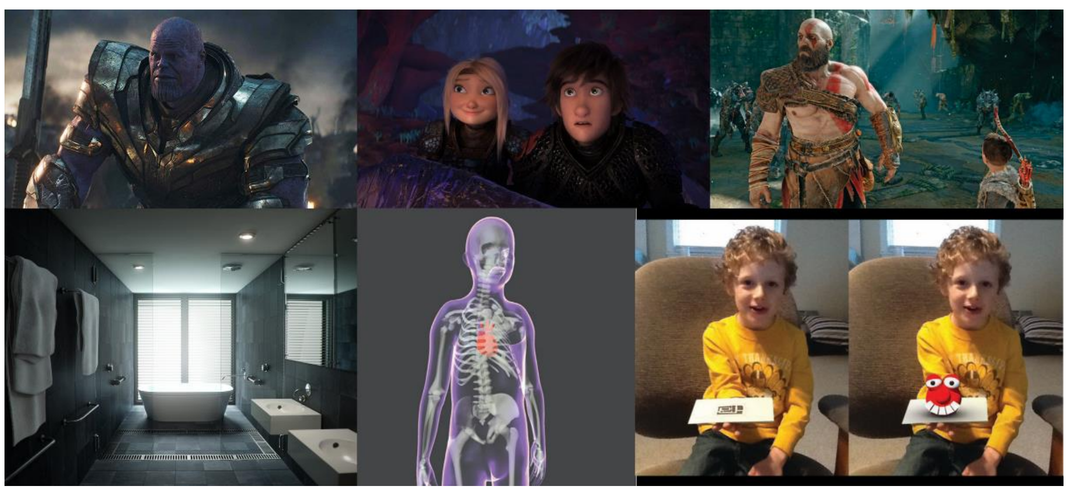
BAB II

TINJAUAN PUSTAKA

2.1. 3D *Animation*

Berawal dari asal katanya, animationem, adalah sebuah teknik untuk menghidupkan benda mati. Awalnya digunakan untuk atraksi sulap, animasi berkembang menjadi sebuah metode untuk penyampaian cerita. Sejak Disney, kata animasi berevolusi menjadi seni gambar bergerak, dibuat dengan *sequence* gambar sebuah aksi berkelanjutan, dan diproyeksikan ke layar dalam kecepatan konstan (Thomas & Johnston, 1995). 3D adalah kependekan dari 3 dimensi, yang merujuk ke ruang kreasi animasi tersebut. Berbeda dengan teknik animasi tradisional yang menggunakan *cel*, animasi ini dibuat dalam ruang tiga dimensi dengan bantuan *software* dan *hardware* komputer (Beane, 2012).

Penggunaan dari animasi 3D sangat luas, meliputi berbagai bidang hiburan, sains dan ilmu pengetahuan, hingga teknologi eksperimental. Animasi 3D paling dikenal aplikasinya dalam film dan animasi, baik itu sebagai *visual effects* dalam film *live-action*, atau sebagai animasi sendiri. Animasi 3D juga sangat dipakai dalam *video games*, visualisasi arsitektural, dan visualisasi medis. Animasi 3D juga menjadi komponen utama dalam pengembangan *virtual reality* dan *augmented reality*.



Gambar 2.1. Beberapa aplikasi dari 3D *animation.* Dari kiri atas, *visual effects* (Russo & Russo, 2019), *animated film* (Deblois, 2019), *video game* (Barlog, 2018), Visualisasi arsitektural (Ranson, 2015), visualisasi medis (Beane, 2012, hlm. 6), dan *augmented reality* (Beane, 2012, hlm. 9)

2.2. *Flowchart*

Dalam sebuah proyek, terdapat tujuan utama yang ingin dicapai. Untuk mencapai tujuan tersebut, dibutuhkan '*work*'. Kata '*work*' didefinisikan sebagai 'sebuah aktivitas yang dilakukan untuk mencapai atau memproduksi sesuatu'. Tentu saja, '*work*' ini harus diatur agar dapat mencapai efisiensi yang tinggi. Muncullah istilah '*flow*', yang berarti 'transformasi *input resource* menjadi hasil akhir sesuai urutan/jalan yang telah ditentukan'. Situasi yang ideal adalah di mana setiap barang dalam tiap langkah dikerjakan tanpa penundaan, cacat atau melalui proses yang mubazir. Keseluruhan dari proses ini disebut dengan '*workflow*'.

Penting bagi organisasi dalam pengerjaan proyek untuk menggambarkan '*workflow*' tersebut. Sebuah penyampaian konsep pekerjaan yang akan dilakukan secara eksplisit akan membantu individu yang membacanya untuk dapat

menggunakan informasi tersebut untuk mencapai hasil yang diinginkan bersama, membuat sebuah pekerjaan yang sebelumnya abstrak menjadi terukur, dan juga meningkatkan kualitas komunikasi, baik itu sesama pekerja atau pekerja dengan klien. Kemudian organisasi dapat menerapkan *Quality Assurance* (QA), integrasi departemen IT, perencanaan strategi, dan manajemen tiap sektor, sehingga meningkatkan efisiensi keseluruhan proyek. (Damelio, 2011)

Untuk dapat menggambarkan '*workflow*', beberapa teknik yang dapat digunakan yaitu *relationship map*, *cross-functional process map* (*swimlane diagram*), dan *flowchart*. Ketiga teknik ini mempunyai kegunaan dan keuntungannya masing-masing. Di sini teknik yang akan digunakan adalah *flowchart*. Berikut adalah beberapa keuntungan pemakaian *flowchart*:

- Menggambarkan suatu bagian kecil dari proses yang lebih besar, untuk mengetahui '*ground truth*', apa yang terjadi di realita.
- Membedakan antara pekerjaan '*value-creating*' dan '*nonvalue-creating*'.
- Membuat pekerjaan '*nonvalue-creating*' menjadi terlihat secara eksplisit.

Flowchart adalah sebuah ilmu yang telah diresmikan dalam ISO (*International Organization for Standardization*). Oleh karena itu, pembuatan dan pembacaannya harus mengikuti *guideline* yang telah disepakati. Berikut adalah tabel simbol *flowchart* yang paling sering dipakai:

Tabel 2.1. Tabel simbol *flowchart*

Simbol	Deskripsi

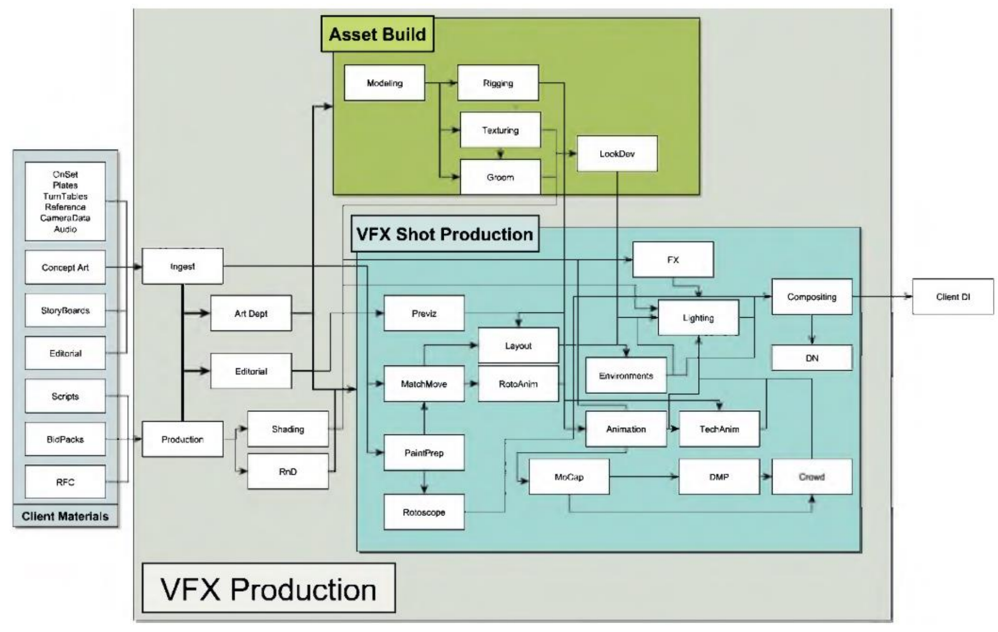
	Terminator, menandakan awal atau akhir sebuah *flowchart.*
	Flow, menunjukkan arah jalan suatu proses.
	Process, melambangkan sebuah proses/pekerjaan. Aktivitas ini membutuhkan *input* dan menghasilkan *output*.
	Decision, memisahkan *flow* sesuai dengan kondisi yang tertera.
	Document, melambangkan keluaran sebuah dokumen.
	On-page connector dan ***off-page connector***, meneruskan *flow* yang terputus akibat kurangnya ruang gambar, dari halaman yang sama atau berbeda.
	Database atau ***magnetic disk***, melambangkan penyimpanan data.
	Data, melambangkan data masukan pengguna atau keluaran yang dihasilkan proses.
	Delay, melambangkan jeda waktu dalam *flow*.

2.3. *Pipeline*

Dalam kamus Merriam-Webster, *pipeline* mempunyai salah satu arti sebagai sebuah system untuk proses *development* ("Pipeline," n.d.). Dalam industri, p*ipeline* adalah sebuah konsep melakukan pekerjaan secara paralel, di mana

pekerjaan yang dilakukan sangat bergantung dengan tahap sebelumnya. Konsep ini dapat diilustrasikan dengan proses pembuatan roti lapis. Pertama-tama, roti akan dipersiapkan. Di atas roti tersebut akan ditambahkan daging, serta bahan-bahan tambahan lainnya. Sebut saja satu tahap membutuhkan waktu 20 detik, maka roti lapis dapat selesai dalam waktu 60 detik. Sebelum satu pekerjaan selesai, hasilnya tidak dapat diberikan ke pekerjaan selanjutnya. Oleh karena itu, pekerjaan yang membutuhkan waktu paling lama sering disebut dengan *bottleneck.* Pekerjaan itu menentukan kecepatan keseluruhan proyek. (Akenine-Möller et al., 2018)

Dunlop (2014) menggambarkan *pipeline* sebagai perekat yang bertanggung jawab untuk menyatukan keseluruhan bagian produksi. *Pipeline* dibentuk untuk menyelesaikan suatu masalah. Dalam kasus ini, masalah yang dihadapi adalah film animasi. Dalam pembentukannya, seorang *pipeline engineer* harus memikirkan cara untuk mem-*breakdown* pekerjaan menjadi lebih sederhana, mendistribusikan tugas untuk para pekerja, dan mengelola data masukan dan keluaran dari setiap proses. Setelah pembuatannya, akan banyak dilakukan iterasi untuk menguji keefektifan *pipeline* tersebut. Sehingga tidak sedikit *pipeline* yang mengalami revisi bahkan ketika proses produksi berjalan.



Gambar 2.2. Contoh *pipeline* sederhana dalam studio VFX

(Dunlop, 2014, hlm. 6)

Untuk mempermudah penyampaian, *pipeline* harus divisualisasikan menggunakan *flowchart*. Seperti yang diperlihatkan dalam gambar 2.1., *pipeline* dibentuk dari jaringan proses yang di dalamnya terdapat jaringan proses yang lebih detail. Setiap proses tersebut melakukan pekerjaannya masing-masing secara berurutan. Dalam setiap proses dibutuhkan data masukan, yang akan dipakai sebagai bahan untuk melakukan proses tersebut, lalu diproses menjadi sebuah data baru. Data keluaran dari proses tersebut lalu dimasukkan ke proses selanjutnya.

Tim produksi akan terdiri dari banyak departemen yang berisi grup-grup dengan pekerjaan yang berbeda. Dari *concept artist* hingga *programmer*, masing-masing berbicara dengan bahasa teknis sendiri. Tidak jarang bagi staf dari departemen berbeda untuk salah berkomunikasi, atau bahkan tidak bisa sama sekali.

Oleh karena itu, *pipeline* harus dapat menjembatani rintangan tersebut. (Dunlop, 2014)

2.3.1. Film dan *Game Development Pipeline*

Walaupun menggunakan teknologi yang mirip, film, baik itu *live action* dengan VFX maupun animasi, dengan gim, mempunyai *pipeline* yang sangat berbeda. Hal pertama yang membedakan adalah keluaran dari kedua media tersebut. Film mempunyai keluaran berupa video yang ditonton oleh audiens. Tidak ada interaksi antara audiens dengan hasil tersebut. Sutradara dan kru pembuatan film mempunyai kontrol penuh dalam penyampaian cerita. Di lain pihak, terdapat interaksi timbal balik antara pemain dengan gim. Pengalaman atau cerita yang hendak disampaikan oleh *developer* berbeda-beda antara satu pemain dengan pemain lain.

Salah satu tantangan terbesar dalam pembuatan pipeline studio film adalah mengelola data dalam ukuran yang sangat besar. Seringkali dalam pembuatan VFX, dibutuhkan keluaran yang *photorealistic*. Untuk me-*render* satu monster, dibutuhkan ratusan bahkan ribuan aset yang harus disatukan. Tidak sedikit kasus di mana *artist* harus memasukkan beberapa *terabyte* data ke dalam render engine. Studio VFX juga harus menyelesaikan permintaan klien dalam waktu 6 bulan – 1 tahun, sehingga membutuhkan tenaga kerja ribuan *artist* dalam satu projek. Studio juga harus mempersiapkan kemungkinan revisi dan perubahan yang disebabkan karena hal-hal tidak terduga. Oleh karena itu, *pipeline* studio film harus dapat mengelola data dengan ukuran besar dengan fleksibel. (Dunlop, 2014)

Berbeda dengan film, gim memberikan kontrol kepada pemain. Sebuah gim harus dapat me-*render* keluaran sesuai dengan masukan dari pemain dari sebuah *controller*. Artinya, sebuah gim harus dapat menyatukan semua aset, baik itu gambar 2D, model 3D, maupun audio, dan me-*render* secara *real-time* agar dapat memberikan pengalaman interaktif yang *seamless*. Oleh karena kebutuhan ini, alat yang digunakan selalu berubah secara drastis dari masa ke masa.

Terlepas dari hal-hal tersebut, tidak ada standar *pipeline* yang harus dipatuhi. Semua studio akan mempunyai *pipeline* mereka masing-masing, bahkan ketika produk keluaran mereka sama secara teknis. (Dunlop, 2014)

2.3.2. Prototipe *Pipeline*

Tahap *pre-production* adalah salah satu tahap yang paling penting dalam menentukan keberlangsungan proyek dalam suatu studio. Dalam tahap ini, ide akan bermunculan banyak. Tidak semua ide-ide tersebut cocok atau bahkan memungkinkan untuk dieksekusi oleh studio. Oleh karena itu, dibutuhkan metode untuk memvalidasi ide-ide tersebut. Di sinilah studio harus membuat prototipe *pipeline* untuk diuji. Prototipe ini adalah *mini production* yang menghasilkan data pengujian *pipeline* baru. (Dunlop, 2014)

2.3.3. Fungsi-fungsi Dasar *Pipeline*

Sebuah *pipeline* dibuat dengan tujuan untuk melakukan produksi dengan efektif dan efisien. Terdapat dua aspek dasar yang harus dicapai dalam melakukan hal tersebut. Pertama, sebuah *pipeline* harus dapat mengatur aliran data dari proses ke proses. Kedua, sebuah *pipeline* juga harus dapat mengarahkan *workflow* dari tugas ke tugas.

Untuk menyelesaikan kedua aspek ini, maka seorang *pipeline engineer* harus mengerti setiap proses di dalam *pipeline* serta mengerti data masukan dan keluaran dari proses-proses tersebut. (Dunlop, 2014, hlm. 97)

Selain pengetahuan dari setiap detil tersebut, seorang *pipeline engineer* juga harus mengerti proyek yang dikerjakan secara keseluruhan. Ukuran dan tingkat kesulitan dari proyek, sumber daya yang tersedia, *tool* yang telah dibuat, bahkan sampai budaya kerja dalam studio. Untuk menentukan *software* yang akan dipakai juga dibutuhkan konsiderasi oleh para *artist*. Jika *software* baru melakukan pekerjaannya dengan cepat, tetapi *artist* dalam studio masih belum familiar, maka penggunaan *software* tersebut bisa saja memakan waktu yang lebih banyak. (Dunlop, 2014, hlm. 98)

Menurut Dunlop, beberapa fungsi dasar sebuah *pipeline* adalah :

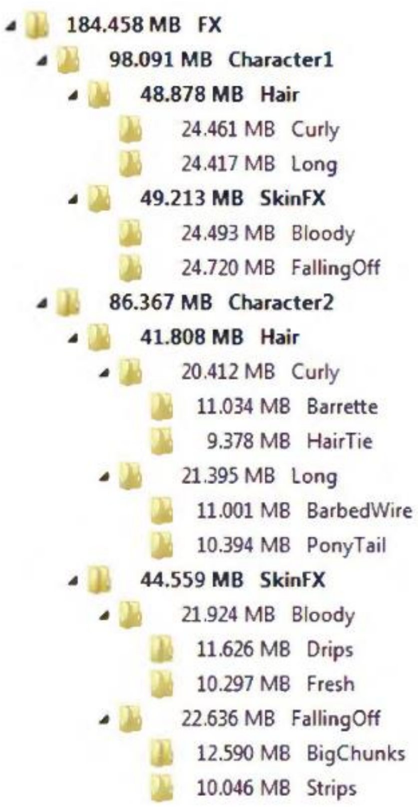
- *File Exchange*
- *Directory Structure*
- *File Naming Convention*
- *Metadata*
- *Versioning*
- *Asset Review and Approval*
- *Production Data Tracking*

2.3.3.1. *File Exchange*

Dalam sebuah proyek, sangat jarang studio hanya memakai satu *software* dari ujung ke ujung *pipeline*. Sebagai contoh, departemen *modeling* bisa saja memakai tiga sekaligus, ZBrush untuk *high-poly sculpting*, xNormal untuk *normal map baking*, dan Maya untuk *retopology* dan penggabungan aset. Seperti pada umumnya, sebuah *software* tidak dapat membaca *file native* dari *software* lain. Oleh karena itu, dibutuhkan format *file* universal yang dapat dibaca oleh *software-software* bersangkutan. Untuk 3D *model* yang sering dipakai adalah .obj dan Filmbox (.fbx). Untuk VFX yang lebih kompleks bisa menggunakan Alembic (.abc). (Dunlop, 2014, hlm. 103)

2.3.3.2. *Directory Structure*

Sebuah proyek VFX maupun *game* bisa mempunyai puluhan ribu *file* yang total ukurannya bisa mencapai beberapa *terabyte*. Setiap *file* tersebut mempunyai kegunaannya masing-masing. Oleh karena itu, dibutuhkan sistem yang dapat mengatur *file-file* tersebut. Salah satunya adalah membuat hierarki *folder* yang rapi dengan membagi-bagi *file* menjadi beberapa segmen. Dunlop (2014) mengungkapkan bahwa pengaturan *file* ini bertujuan untuk menyembunyikan apa yang tidak boleh dilihat, daripada menemukan *file* spesifik.



Gambar 2.3. Contoh *directory structure*

(Dunlop, 2014, hlm. 187)

Pembagian *folder-folder* adalah berdasarkan fungsionalitas dari isinya. Walaupun begitu, setiap departemen dalam sebuah studio pasti memiliki cara pembagiannya masing-masing. Departemen *modeler* menginginkan *folder*-nya dibagi berdasarkan ras seperti *elf*, *orc*, *human*, dsb. Di lain pihak, departemen animasi menginginkan pembagian berdasarkan *gender* agar animasinya dapat dipakai ulang sesuai *gender* karakternya. Muncullah departemen *game designer* yang ingin pembagian berdasarkan *playable* dan *non-playable*. Argumen-argumen seperti ini sangat sering terjadi di industri, dan tidak dapat dihindari. Untuk memperparah keadaan,

pembagian ini seperti pembagian wilayah kekuasaan. Di mana ada kekuasaan, pasti ada perang.

Penyelesaiannya adalah dengan memilih salah satu jalur, dan mengikuti aturannya hingga akhir. Tentu saja beberapa departemen harus melakukan kompromi. Keputusan tersebut harus didokumentasikan alasannya, sehingga anggota baru dapat mengikutinya dengan efektif. (Dunlop, 2014, hlm. 106)

Sebuah *directory structure* dapat berupa *flat* atau *deep*. *Deep* berarti strukturnya mempunyai banyak *sub-folder*, dengan konten individu tiap *folder* yang sedikit. *Flat* berarti *sub-folder* yang sedikit, tetapi konten yang banyak untuk tiap *folder*. *Directory structure* harus di-desain secara seimbang untuk tidak terlalu dalam sehingga pengguna dapat tersesat, tetapi juga tidak terlalu dangkal sehingga memiliki terlalu banyak *file* dalam satu *folder*.

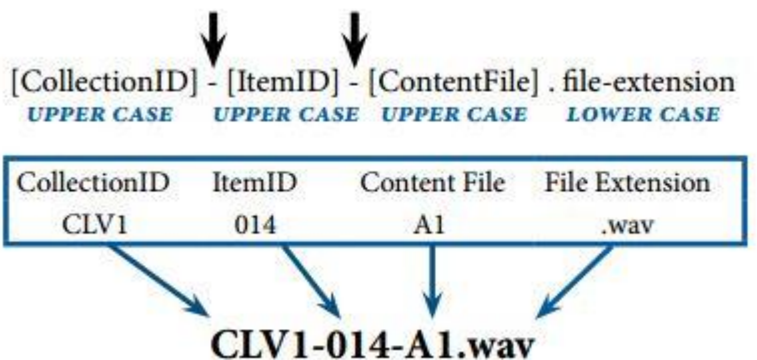
Oleh karena penggunanya yang kebanyakan manusia, *directory structure* juga harus di-desain dengan kemudahan navigasi sebagai salah satu faktor utamanya. Ini berarti menganalisis *workflow* dari *artist* dan merancang struktur yang mudah untuk ditelusuri dan *step* yang minimal untuk berpindah dari *folder* ke *folder* lain.

Untuk membangun pembagian *folder* yang baik, Dunlop menyarankan untuk membagi aset dari *generic* hingga *specific*. Tentu saja, definisi dari *generic* dapat berbeda dari *project* ke *project*. Sebagai contoh, sebuah film *zombie* membutuhkan para *artist*-nya untuk membuat FX untuk

kulit dan rambut dari karakternya. Metode pertama adalah membagi berdasarkan karakter sebagai *generic*. Ini berarti setiap karakter mempunyai FX yang unik. Metode kedua adalah membagi berdasarkan tipe FX. Setiap FX dengan tipe sama akan dibedakan sedikit untuk setiap karakter.

2.3.3.3. *File Naming Convention*

Setelah struktur *folder* yang baik, dibutuhkan penamaan *file* yang baik pula. Sebuah penamaan yang baik dapat dibaca oleh siapapun yang ada di dalam proyek tersebut, dan dapat langsung dimengerti dari mana asalnya, dan apa kegunaannya. Dengan kata lain, dapat diidentifikasi secara global. Dengan penamaan yang baik, proses pencarian data dapat dilakukan bahkan secara manual oleh *artist*. Beberapa studio membuat *tool* khusus untuk membuat penamaan menjadi rapi. Selain penamaan *file*, menggunakan format yang benar untuk sebuah *file* juga penting. (Dunlop, 2014, hlm. 108)



Gambar 2.4. Contoh *naming convention*

(“File Naming Conventions,” n.d.)

Untuk memisahkan elemen-elemen dalam sebuah nama, dapat digunakan *underscore*, ataupun menggunakan huruf kapital. Akan tetapi, tidak dianjurkan untuk menggunakan spasi. Walaupun berfungsi pada 90% *software*, spasi akan memberikan *error* yang tidak terduga pada sisanya. Contoh : “sword_steel_large.extension” dan “swordSteelLarge.extension”

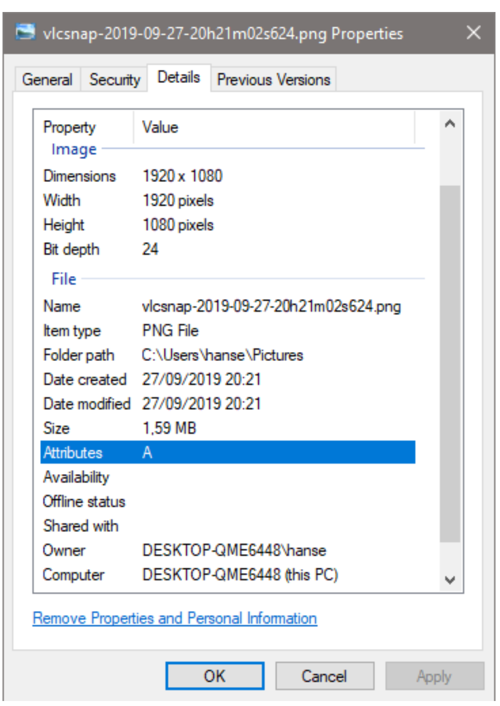
Penggunaan huruf kapital dapat menjadi masalah pada *tool* atau *script* yang metode pencariannya adalah *case sensitive*. Cara termudah untuk menghindarinya adalah menggunakan *convention* yang konsisten dari awal hingga akhir *project*.

Penggunaan singkatan juga harus dibatasi. Walaupun singkatan dapat memangkas waktu penulisan nama *file*, penggunaannya dapat membingungkan pengguna lain apabila tidak didokumentasikan dengan baik dan tidak digunakan secara konsisten.

Salah satu *convention* yang sering dipakai adalah pencerminan letak dari *file* dalam struktur *folder*. Nama-nama dari *folder* ditulis secara berurutan pada nama *file*. Contoh : sebuah *file* dalam *folder* /GameRoot/Assets/Models/Equipment/Weapon/Sword/Rare/ dapat bernama “eq_wp_sword_rare_demonshard_a.mdl” atau “eqWpSwordRareDemonshardA.mdl”. Penggunaan metode ini dapat membantu kerja *scripting*, dan juga membantu mencegah penamaan *file* yang sama.

2.3.3.4. Metadata

Untuk mengatur data, banyak cara dapat dilakukan selain mengatur *folder* di mana mereka ditempatkan. Salah satunya adalah metadata. Metadata menampung informasi seperti nama pembuat, tanggal dibuat, *copyright*, *genre* dan album (dalam musik), dsb. tergantung dengan kebutuhan. Data-data tersebut dapat digunakan sebagai *tag* yang disaring oleh *filter* ketika melakukan *search*. Dibutuhkan sebuah *custom tool* untuk para *artist* agar dapat membuat metadata yang rapi dan siap dipakai. (Dunlop, 2014, hlm. 111)



Gambar 2.5. *Metadata* sebuah gambar

Dokumentasi Pribadi

2.3.3.5. *Versioning*

Pengerjaan sebuah proyek adalah proses yang memakan waktu dan penuh dengan revisi. Satu *file* aset tidak hanya dibuat sekali jadi dan dapat dipakai hingga akhir proyek. *File* tersebut akan menerima banyak revisi baik dari supervisi maupun klien. Proses revisi juga dilakukan oleh banyak orang. Oleh karena itu, dibutuhkan sistem *versioning*. *Versioning* tidak hanya berarti pemberian nomor versi pada *file*.

Versioning dilakukan dengan tujuan utama memisahkan pekerjaan agar tidak ada dua orang yang mengerjakan satu *file* dalam waktu bersamaan. Dalam *software* seperti Perforce, Shotgun, atau Subversion, anggota dalam studio dapat melakukan *check-out* dan *check-in*. Ketika *check-out*, *file* dari server akan di-*copy* ke penyimpanan lokal, lalu *file* yang ada di server akan dikunci sehingga orang lain tidak dapat melakukan *check-out*. Setelah melakukan pekerjaannya, *check-in* akan dilakukan, yaitu memasukkan *file* yang sudah diubah ke dalam server. (Dunlop, 2014, hlm. 115)

Name	Date modified	Type	Size
[CHR01]_Jono Rigging 00.ma	12/09/2019 21:16	Maya ASCII File	1.847 KB
[CHR01]_Jono Rigging 01.ma	12/09/2019 21:16	Maya ASCII File	5.852 KB
[CHR01]_Jono Rigging 02.ma	12/09/2019 21:15	Maya ASCII File	22.195 KB
[CHR01]_Jono Rigging 03.ma	12/09/2019 21:15	Maya ASCII File	16.821 KB
[CHR01]_Jono Rigging 04.ma	12/09/2019 21:15	Maya ASCII File	16.843 KB
[CHR01]_Jono Rigging 05.ma	12/09/2019 21:15	Maya ASCII File	16.940 KB
[CHR01]_Jono Rigging 06.ma	12/09/2019 21:15	Maya ASCII File	16.892 KB
[CHR01]_Jono Rigging 07.ma	12/09/2019 21:05	Maya ASCII File	16.908 KB
[CHR01]_Jono Rigging 08.ma	12/09/2019 21:05	Maya ASCII File	17.161 KB
[CHR01]_Jono Rigging 09.ma	12/09/2019 21:15	Maya ASCII File	17.167 KB
[CHR01]_Jono Rigging 10.ma	11/09/2019 22:03	Maya ASCII File	19.286 KB
[CHR01]_Jono Rigging 11.ma	12/09/2019 8:05	Maya ASCII File	19.326 KB
[CHR01]_Jono Rigging 12.ma	12/09/2019 8:49	Maya ASCII File	19.694 KB
[CHR01]_Jono Rigging 13.ma	13/09/2019 0:15	Maya ASCII File	17.002 KB
[CHR01]_Jono Rigging 14.ma	13/09/2019 1:06	Maya ASCII File	16.997 KB
[CHR01]_Jono Rigging 15.ma	13/09/2019 1:47	Maya ASCII File	16.983 KB
[CHR01]_Jono Rigging 16.ma	13/09/2019 1:53	Maya ASCII File	17.019 KB
[CHR01]_Jono Rigging 17.ma	13/09/2019 2:25	Maya ASCII File	17.651 KB
[CHR01]_Jono Rigging 18.ma	13/09/2019 2:59	Maya ASCII File	19.756 KB

Gambar 2.6. Contoh *Versioning*

Dokumentasi Pribadi

2.3.3.6. *Asset Review and Approval*

Pekerjaan dari *artist* harus di-*review* oleh supervisi dan klien. *Review* ini dapat dilakukan secara informal, seperti melihat *over-the-shoulder* langsung di *workstation* maupun formal, seperti *review* harian yang dilakukan oleh komite khusus. Dalam *review* tersebut akan dibuat catatan revisi yang harus diberi ke departemen yang bersangkutan. Hasil yang direvisi tidak pernah disebut sebagai *final*, melainkan *could be better*, yang berarti sudah cukup untuk konsumsi publik, namun dapat diperbagus jika waktu memungkinkan. (Dunlop, 2014, hlm. 121)

Penilaian *shot* tidak dapat dilakukan secara individu. Ini dikarenakan *shot* tidak dapat berdiri sendiri, harus disertai oleh *shot-shot* lain untuk memperlihatkan sebuah kontinuitas. Oleh karena itu, biasanya

studio mengerjakan *shot* sampai 75%, lalu me-*review* semua *shot*. Pendekatan ini lebih efektif dibandingkan menyempurnakan satu *shot* baru beralih ke *shot* selanjutnya. Revisi dalam satu *shot* juga dapat mempengaruhi *shot* lain, sehingga tidak membuang waktu untuk tahap *polishing*. (Dunlop, 2014)

2.3.3.7. *Production Data Tracking*

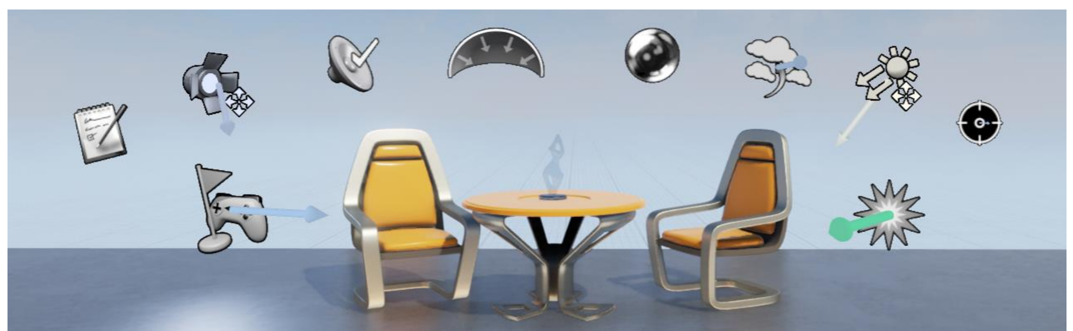
Dalam proses produksi, terdapat data yang harus di-*track* setiap saat. Data-data tersebut menjelaskan tentang status sementara dari proyek tersebut. Contohnya seperti, siapa sedang mengerjakan apa, *deadline* dari pekerjaan tersebut, apa yang sedang di-*review*, penyelesaian revisi, dan sudah sedekat apa dengan tujuan. Pengambilan data tersebut bisa dari *input* manual, maupun secara otomatis melalui sebuah *tool* khusus. Adanya data-data tersebut dapat membantu dalam pengambilan keputusan di kemudian hari. (Dunlop, 2014, hlm. 123)

2.4. Unreal Engine 4

Unreal Engine adalah sebuah *game engine* yang awalnya ditulis oleh Tim Sweeney untuk mengembangkan proyek *game first person shooter* “Unreal” pada tahun 1998. Sekarang Unreal Engine dikembangkan oleh Epic Games, dengan versi paling barunya, Unreal Engine 4, atau biasa disingkat UE4. Seiring dengan perkembangan teknologi, UE4 mengembangkan teknologi *ray tracing*. Berikut adalah penjelasan singkat mengenai fitur-fitur dalam UE4.

2.4.1. *Actor*

Dalam UE4, *actor* adalah objek apapun yang dimasukkan ke dalam *level*. Secara teknis, *actor* adalah tipe *Class* yang dapat menampung informasi transformasi. Beberapa contoh meliputi *geometry*, *player*, *camera*, bahkan *script* ("Actors and Geometry," n.d.). *Class* adalah sebuah konsep dalam *programming* yang dapat digunakan untuk membuat objek yang mempunyai fitur *built-in*. Dari sini, *programmer* dapat membuat *Class* turunan dari *template* tersebut, dan memanfaatkan relasi antar *Class* (Stroustrup, 1997, hlm. 223).



Gambar 2.7. Macam-macam *actor* dalam UE4

(https://docs.unrealengine.com/en-us/Engine/Actors, n.d.)

2.4.1.1. Tipe-tipe *Actor*

Berikut ini adalah tipe-tipe *actor* yang harus diketahui dalam penggunaan Unreal Engine 4:

StaticMeshActor adalah tipe *actor* yang berfungsi untuk menampilkan *mesh* dalam *scene*. *Mesh* yang ditampilkan adalah berupa *static mesh*, geometri yang

tidak akan berubah selama berjalannya gim. Contoh penggunaannya seperti *environment* dan dekorasi.

Brush adalah tipe *actor* yang berfungsi untuk menampilkan geometri sederhana yang dapat diedit secara langsung dalam *editor*. Kegunaan utamanya adalah untuk memvisualisasikan suatu *environment* atau membuat prototip *gameplay* dalam waktu yang cepat.

SkeletalMeshActor adalah tipe *actor* yang berfungsi untuk menampilkan *skeletal mesh*, geometri yang dapat di-*deform* dengan animasi yang di-*export* melalui aplikasi 3D. Karakter yang bergerak secara kompleks, maupun mesin yang canggih dapat menggunakan *actor* tipe ini. (“Common Actor Types,” n.d.)

CameraActor adalah tipe *actor* yang berfungsi untuk melihat dan merekam gambar untuk diperlihatkan ke *player*. *CameraActor* dapat digunakan untuk keperluan dalam *gameplay* dengan menghubungkannya ke *blueprint* yang bersangkutan (“Camera Actors,” n.d.). Terdapat juga *CineCameraActor*, yang mempunyai fungsionalitas layaknya kamera di dunia nyata (“Using Cine Camera Actors,” n.d.). *Actor* ini berfungsi untuk mempermudah pembuatan *cinematic*.

PointLight, *SpotLight*, *DirectionalLight*, *SkyLight*, dan *RectLight* adalah tipe-tipe *actor* yang mensimulasikan cahaya. *PointLight* mengeluarkan cahaya dari satu titik ke semua arah, seperti lampu bohlam. *SpotLight* mengeluarkan cahaya dari satu titik, tetapi dibatasi dalam sebuah *cone*. *DirectionalLight* mengeluarkan cahaya dari tempat yang mendekati *infinite*, dan mempunyai arah cahaya yang sama. Cahaya ini biasa digunakan untuk adegan *outdoor*. *SkyLight* mengeluarkan cahaya

sesuai dengan *background* yang di-*capture* secara *real-time*, atau dengan gambar HDRI. *RectLight*, sesuai namanya, mengeluarkan cahaya dalam bentuk persegi panjang. (“Types of Lights,” n.d.)

2.4.1.2. *Actor Mobility*

Transform			
Location	X -215.0	Y -120.0	Z 32.0
Rotation	X -0.0 °	Y 0.0 °	Z 63.749863 °
Scale	X 1.0	Y 1.0	Z 1.0
Mobility	Static	Stationary	Movable

Gambar 2.8. Tiga Pilihan *Actor Mobility*

(“Actor Mobility,” n.d.)

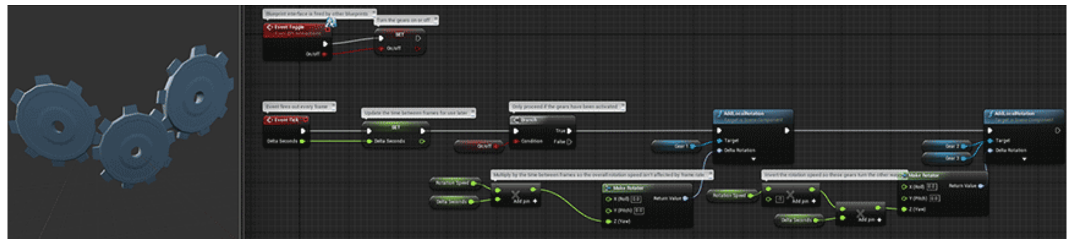
UE4 membagi *actor* menjadi tiga kategori sesuai dengan *mobility*-nya dalam *scene*. *Static* berarti *actor* tidak akan bergerak sama sekali dan bersifat permanen dalam *scene*. Untuk memaksimalkan waktu render, UE4 akan melakukan *light baking* ke objek-objek tersebut. *Actor* yang berupa *light* akan berkontribusi penuh ke *light map*. *Stationary* berarti walaupun *actor* tidak akan bergerak, tetapi dapat diganti di tengah *runtime*. Objek-objek tersebut tidak akan berkontribusi ke *light map*, tetapi akan menggunakan *cached shadow* ketika diterangi oleh *movable light*. *Movable* berarti *actor* akan bergerak secara bebas dalam *scene*. Objek-objek *movable* tidak akan berkontribusi ke *light map* sama sekali, yang berarti adalah objek paling berat untuk di-*render*. (“Actor Mobility,” n.d.)

2.4.2. Level

Karena UE4 awalnya ditulis untuk *environment* pengembangan gim, maka terdapat istilah *level*. *Level* adalah tempat di mana semua *actor*, seperti *static mesh*, *blueprint*, dan cahaya, ditempatkan dalam sebuah *container*. Ukurannya bisa seluas padang rumput ataupun sekecil ruangan ("Levels," n.d.). Di dalam sebuah *level* terdapat *persistent level*, yang berperan sebagai dasar. Apapun yang ditempatkan di *persistent level* akan muncul di setiap *sublevel*. Biasanya *persistent level* dibiarkan kosong, baru di bawahnya dimasukkan *sublevel* yang berupa *environment*, karakter, *lighting*, *cine*, *blueprint*, dan *post-processing*. Setiap *sublevel* tersebut dapat dimatikan atau dinyalakan kapanpun, sehingga fitur ini mirip dengan fitur *layer* dalam Maya. (*Fortnite Trailer: Developing a real-time pipeline for a faster workflow (White Paper)*, n.d., hlm. 19)

Gerakkan dan animasi dalam sebuah *level* dimasukkan sebagai *track*. *Track* dapat menampung animasi, transformasi, ataupun suara. *Level sequence* juga dapat dimasukkan ke dalam *level sequence* lain. Dengan fleksibilitas ini, *filmmaker* dapat membuat *pipeline* yang sesuai dengan studio. (*Fortnite Trailer: Developing a real-time pipeline for a faster workflow (White Paper)*, n.d., hlm. 19)

2.4.3. *Blueprint*

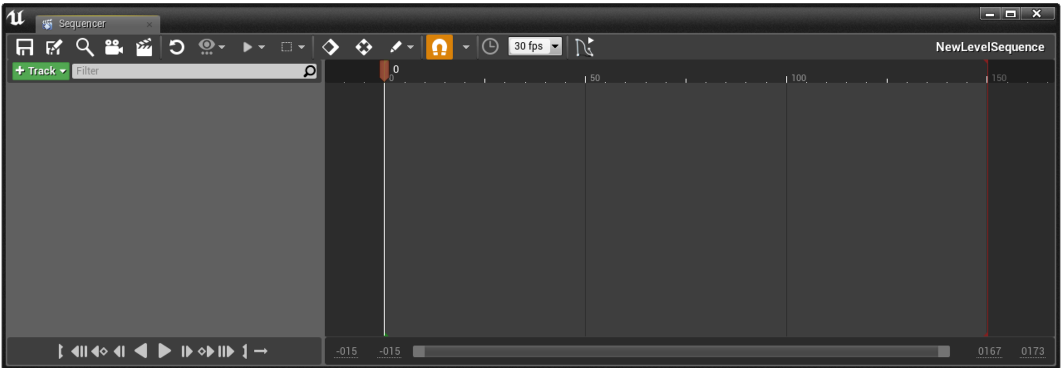


Gambar 2.9. *Blueprint*, sistem *visual scripting* UE4

("Blueprints," n.d.)

Blueprint adalah sistem *visual scripting* yang dipakai dalam UE4. Seperti bahasa pemrograman biasa, *blueprint* dapat digunakan untuk melakukan *object-oriented programming*. Contohnya seperti mendefinisikan *class* atau *object* yang dapat dipakai nantinya. Dengan sistem ini, *tool-tool* yang hanya tersedia untuk *programmer* tersedia juga untuk *designer* dan *artist*. ("Blueprints," n.d.)

2.4.4. Sequencer



Gambar 2.10. *Sequencer* dalam UE4

("Sequencer Overview," n.d.)

Sequencer adalah fitur dalam UE4 untuk membuat *in-game cinematics*. *Artist* dapat memasukkan *sequence* dan *track* ke dalamnya. *Track* adalah *keyframe* yang dapat mengatur hal-hal seperti animasi untuk karakter, transformasi untuk menempatkan *actor* dalam *scene*, *visibility*, atau bahkan audio ("Sequencer Overview," n.d.). Setiap *sequence* animasi yang telah di-*breakdown* dalam proses pra-produksi di-*import* ke dalam *sequencer* sebagai sebuah *level*. Setiap *level* tersebut dapat di-*review* dan di-edit sebagai satu kesatuan tanpa menghilangkan akses ke data dan aset dari *level* lain. (*Fortnite Trailer: Developing a real-time pipeline for a faster workflow (White Paper)*, n.d., hlm. 18)

2.4.5. *File Exchange*

Untuk memulai proses produksi, para *artist* mulai melakukan *asset creation* menggunakan *software* 3D seperti Autodesk 3DS Max, Autodesk Maya, ataupun Pixologic ZBrush. Aset-aset yang telah dibuat tentunya menggunakan format asli dari *software* tersebut. 3DS Max menggunakan .max, Maya menggunakan .ma, dan Zbrush menggunakan .zpr. Unreal Engine tidak dapat membaca aset tersebut secara langsung. Oleh karena itu, dibutuhkan format untuk *file exchange*.

Filmbox (FBX)

FBX adalah format *file* yang dipunyai oleh Autodesk. Format ini dibuat khusus untuk pemindahan data antar *software* seperti 3DS Max, Maya, MotionBuilder. Unreal dapat melakukan *import* terhadap file FBX. Dengan format ini, Unreal dapat membaca: *Static mesh*, *skeletal mesh*, animasi, *morph target*, LOD (*Level of Detail*),

material dan *texture*. Selain itu, beberapa aset juga dapat dimasukkan ke dalam satu *file* FBX. (“FBX Content Pipeline,” n.d.)

Aset karakter akan diklasifikasikan menjadi *skeletal mesh* dalam UE4. Dalam file FBX karakter tersebut terdapat 3 hal yang paling utama. Pertama adalah *geometry* dari model karakter. Kedua adalah hierarki *joint* yang digunakan sebagai *skeleton* dari model tersebut. Terakhir adalah data *skinning*, yang membuat *geometry* terpengaruh oleh gerakan *joint*. Data lain yang dapat dimasukkan meliputi *smoothing group*, LOD, *material*, *texture*, dan *vertex color* sesuai dengan kebutuhan. (“FBX Skeletal Mesh Pipeline,” n.d.)

Aset yang tidak bergerak dalam *scene*, diklasifikasikan menjadi *static mesh* dalam UE4. Proses *export* sebenarnya cukup mudah untuk dilakukan. Walaupun begitu ada beberapa hal yang perlu diperhatikan. Pivot akan diletakkan di koordinat (0,0,0) dalam UE4, sehingga *artist* perlu untuk mengatur *mesh* di letak yang seharusnya. Seperti semua *game engine* lainnya, UE4 hanya dapat menerima *geometry* yang terdiri dari *tris*, bukan *quad*. Oleh karena itu, baik *static* maupun *skeletal mesh*, ada baiknya untuk melakukan *triangulation* secara manual sebelum melakukan *export* untuk kontrol penuh. (“FBX Static Mesh Pipeline,” n.d.)

Animasi harus di-*export* secara individual, satu *skeleton* untuk satu *file*. Hanya *skeleton* yang diperlukan ketika melakukan *export*. Format penamaan *file* harus benar. Setelah di-*import* ke dalam UE4, animasi tersebut dapat digunakan untuk karakter-karakter yang menggunakan *skeleton* yang sama. Dengan begitu,

pipeline animasi dapat berjalan lebih efisien untuk karakter yang mirip secara gerakan. (“FBX Animation Pipeline,” n.d.)

Alembic (ABC)

Walaupun FBX dapat digunakan untuk *file exchange*, format tersebut mempunyai banyak batasan dan sangat sulit untuk ditebak. Untuk memecahkan masalah tersebut, Lucasfilm dan Sony Pictures Imageworks mengembangkan sebuah format *open-source* baru, yaitu Alembic. Format ini sangat baik digunakan untuk mengirim data animasi kompleks dalam bentuk *baked geometry*. Format ini mulai diadopsi secara internasional (Dunlop, 2014, hlm. 104). Unreal Engine 4 mulai menerapkan *pipeline* untuk format jenis ini. Fitur ini masih tergolong *experimental* (“Alembic File Importer,” n.d.).