A 3D Modeling System based on Polar Meshes

Daniele Brazzolotto

Kongens Lyngby 2013
COMPUTE-M.Sc-2013-63

Supervisors:
DTU: J. Andreas Bærenten - janba@dtu.dk
Padova: Giovanni De Poli - depoli@dei.unipd.it

Technical University of Denmark
Applied Mathematics and Computer Science
Building 303B, DK-2800 Kongens Lyngby, Denmark
Phone +45 4525 3031, Fax +45 4588 1399
compute@compute.dtu.dk
www.compute.dtu.dk Compute-M.Sc-2013-63

Summary (English)

In 3D multimedia productions, modeling is an activity that absorbs a considerable amount of effort and during the years it has been the focus of an entire line of research. This research field includes SQM, a method originally developed by J. A. Bærentzen, M. K. Misztal and K. Welnicka [JAB12], where the final mesh is procedurally inferred from the skeleton that the designer has modeled. It turned out to be a great tool to quickly define the high level structure of the object but it is strongly limited in terms of mesh details, the exact opposite of a traditional 3D tool.

This project aims to design a new modeling system that merges the benefits of SQM with the strengths of a traditional sculpturing tool, recognizing the needs for two different modeling levels: at high level, polar meshes have been used to quickly define the structure of the object, while at low level the designer is free to add mesh details like in any other 3D application.

In this thesis a core set of polar mesh operations have been defined and implemented in a small modeling prototype. On top of that, it has also been developed a behavior based animation engine and an L-System, to evaluate the benefits that a modeling system based on polar meshes can offer to these areas.

The new modeling system pushes the production of 3D assets towards a more AGILE process, it can procedurally recognize and generate the mesh skeleton and it naively supports structure based operations, including morphing, bottom-up animations and procedurally generated content.

Summary (Italian)

Nella produzione di contenuti multimediali, la modellazione di asset 3D è un'attività che richiede una consistente porzione del costo complessivo di sviluppo ed è al centro di un intero filone di ricerca che include SQM, un metodo sviluppato da J. A. Bærentzen, M. K. Misztal e K. Welnicka [JAB12] in cui la mesh viene automaticamente generata dall'ossatura indicata dal designer. Si tratta di un metodo estremamente efficace per definire la struttura ad alto livello del modello, ma che è anche fortemente limitato per quanto riguarda i particolari che è possibile aggiungere alla mesh stessa, l'esatto opposto del tipico software di modellazione tradizionale.

L'obiettivo di questo lavoro è la progettazione di un sistema di modellazione che sia in grado di unire i vantaggi di SQM con i punti di forza dei tradizionali software 3D, riconoscendo la necessità di due distinti livelli operativi: ad alto livello verranno sfruttate le proprietà delle mesh polari per definire la struttura di base dell'oggetto, mentre a basso livello il designer sarà libero di aggiungere particolari alla mesh come in ogni altro software 3D.

In questa tesi sono state progettate un insieme di operazioni per mesh polari, poi implementate in un piccolo tool di modellazione. E' stato inoltre sviluppato un motore di animazione comportamentale e un L-System per valutare i benefici che un sistema di modellazione basato su mesh polari può offrire anche in riferimento a questi ambiti.

Il nuovo sistema di modellazione spinge la produzione di elementi 3D verso un processo di sviluppo più AGILE, è in grado di generare automaticamente la struttura della mesh e supporta nativamente operazioni come morphing, animationi bottom-up e la generazione automatica di asset multimediali.

Preface

This thesis was prepared at DTU Compute, Department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring a M.Sc. in Digital Media Engineering as well as a M.Sc. in Computer Engineering, respectively at the Technical University of Denmark and at the University of Padova, according to the bilateral agreements that regulate the collaboration between these institutions in the frame of the T.I.M.E. double degree program (https://www.time-association.org/).

This project represents an evolution of the SQM method which has been originally developed by J. A. Bærentzen, M. K. Misztal and K. Welnicka [JAB12] and that has been later analyzed and extended by Michael Mc Donnell [Don12]. SQM is part of a more general line of research that aims to provide a better support for 3D mesh manipulation not only from the point of view of modeling but also for animations and procedurally generated content. In this framework, more abstract tools can contribute to reduce the cost of 3D digital assets, increasing the productivity of designers and animators.

In his thesis, Mc Donnell proved that SQM is affected by severe limitations when the object which needs to be modeled is inorganic or it hasn't a clear structure. These limitations are more an intrinsic characteristic of the modeling system rather than a consequence of the limited set of node types of the original SQM method.

This project proposes a new modeling system, based on the same mesh topology that characterizes SQM, but it has been developed from a completely different set of principles: the mesh is a numerical approximation of the smooth surface that the designer wants to model, it includes structural information regarding the object which has been represented as well as surface details, but defining what is structure

and what is detail is a pure design choice. An effective modeling system maintains a balance between those two layers and it supports both of them with different, although integrated, sets of mesh tools.

The modeling system that will be presented in this report is the result of an iterative process that has redefined multiple times the characteristics of the desired tool, led by a progressively deeper understanding of the of the mesh topology which ultimately has affected also the concept of polar mesh itself.

This thesis doesn't provide a complete modeling tool, neither it is an exhaustive exploration of opportunities and limitations offered by this approach, but it does provide an overview of the potential that a modeling system based on polar meshes can offer. In this perspective this project yielded a positive outcome, it is definitely a proof of concept that motivates further developments, both academically and commercially, and there are the basics for a first integration with existing production tools.

Lyngby, 15-July-2013

Daniele Brazzolotto
daniele.brazzolotto@gmail.com
http://linkedin.com/in/danielebrazzolotto

Acknowledgments

First of all I would like to thank my family for its constant support since the very beginning of my Danish adventure, when I decided to apply for this double degree program. The last two years represented an incredible opportunity to grow up personally and professionally in a way that I couldn't have imagined before. I'm thankful to them all for giving me this opportunity and for supporting me throughout the challenges that characterized this intense period of my life.

I would also like to thank my supervisor J. A. Bærentzen for his assistance during all the phases of this thesis work as well as for providing the GEL framework, which considerably simplified the technical implementation of the prototype. I show my appreciation to DTU-Compute for hosting this project and specifically to Jeppe Revall Frisvad for providing, together with J. A. Bærentzen, weekly feedback throughout the entire working process.

As double degree T.I.M.E. student (https://www.time-association.org/), I would like to show my appreciation to my italian supervisor Giovanni De Poli for his support and finally I would like to thank the Technical University of Denmark and the University of Padova, I'm grateful to whoever in these institutions maintains and improves the T.I.M.E network, promoting the mutual academic recognition.

Contents

Chapter 1

Introduction

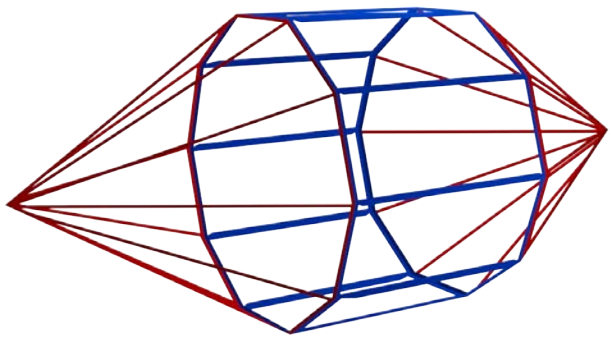


Figure 1.1: Polar Mesh: example with an annular region (blue) surrounded by two polar regions (red)

1.1 Prerequisites

In order to understand the frame that generates and supports this project, it is important to have some preliminary knowledge on basic modeling concepts.

The core of this thesis is the development of a modeling system, a collection of technologies, tools and techniques that designers can use to produce 3D assets. In other words a modeling system is the set of principles that are at the base of a 3D modeling software and which are indirectly shown through the operations that the tool makes available to the user.

A second key concept for this project is polar meshes, polygonal meshes characterized by a strictly regulated topology that supports the polar subdivision scheme [KP07b] and that can be seen as a collection of polar and annular regions [KP07a], or more informally as a collection of cones and cylinders (figure 1.1).

A fundamental property of polar meshes is that it is possible to procedurally infer their skeleton, an abstract representation of the structure of the mesh, where the object has been collapsed into a set of connected segments with the same branching structure of the original model. This operation isn't always easy for general meshes, but this particular topology makes it fast and convenient: cones and cylinders in the mesh can be easily and intuitively collapsed into skeleton segments between the top and the base of these individual elements, while the mesh connectivity defines the way these bones are linked together to form the complete skeleton.

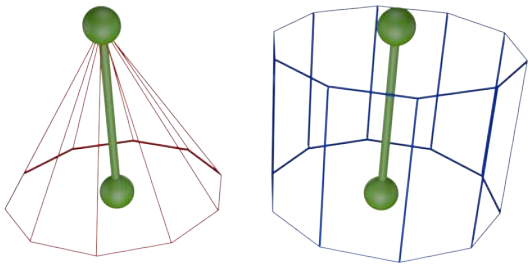


Figure 1.2: Polar Mesh: example of skeleton recognition for polar (red) and annular (blue) regions

1.2 Purposes

In 3D multimedia productions, modeling is an activity that absorbs a considerable portion of the overall effort: for example, according to a recent analysis [Sta, Ros] and direct experience, in the development of a typical 3D video-game, the art

department absorbs 40% to 75% of the entire production manpower. Modeling, rigging[1] and texturing[2] the few organic objects requires 60% to 90% of the resources of the art team and it affects 30% to 65% of the entire production cost. Therefore a 30% optimization of the modeling phase will directly reduce the production cost of 10% - 20%.

This project is part of a more general line of research that aims to speed-up and simplify the modeling of complex objects, providing a better support for 3D mesh manipulation. In this work, this goal will be reached defining and implementing a set of mesh operations that will positively affect the modeling phase as well as animations and procedurally generated content.

Nowadays the main problem with 3D modeling is the low level of abstraction: most of the 3D tools allow the designers to work with vertices and faces, but nothing more than that. It is like to construct a building with bricks and mortar: it surely can be done, but prefabricated components can make the work faster, cheaper and easier.

The idea behind this thesis is to exploit some of the characteristics of polar meshes to give to designers the right tools to build and use the virtual equivalents of prefabricated concrete components:

Skeleton: every polar mesh has an implicit skeleton, that can be procedurally inferred run time. This isn't only suitable to automatically generate a bone structure for future animations, but it is also a way to group vertices and faces together. This is a first abstraction level to design operations that affect several faces at once in a meaningful and coherent way.

Features: in a typical polar mesh, most of the skeleton segments are simply placed in a line, without any branching. A consecutive, connected sequence of skeleton segments can be grouped in a *feature*. This is a second important abstraction level for interesting mesh operations, that get closer to how human brain gives meaning to things: if you look at figure 1.3, you will have no doubt that it is a little man thanks mainly to the branching structure of the lines. Mesh operations at feature level reflect the way a designer thinks and means and, for this reason, they can play an important role in tools that aim to be effective and easy to use.

[1] Rigging: http://en.wikipedia.org/wiki/Skeletal_animation
[2] Texturing or texture mapping: http://en.wikipedia.org/wiki/Texture_mapping

Figure 1.3: Stylized man: example of human brain's capability to recognize shapes based on branching structure

1.3 Expected outcomes

The benefits of the improved modeling system and of the higher abstraction level not only cover different areas of the 3D modeling pipeline but they are also the basics for new real time techniques.

1.3.1 Faster modeling

A first important expected outcome is surely a faster modeling system: as it has been previously discussed, modeling time can be significantly reduced with feature based tools and operations. This single improvement would be enough to justify the entire thesis work as it has a direct impact on the costs of most 3D productions.

1.3.2 Agile modeling

A lesson that every software developer quickly learns is that a software (or a digital product in general) is always a unique piece. Probably a similar need can be required in the future, but the exact same need is very unlikely; therefore one can surely use a previous work, but it must be changed and customized to the new request.

This fuzzy concept of re-usage is well known in the world of software development as well as in some other areas, but unfortunately the production of 3D assets is far behind on this process: ask a 3D artist to change the structure of a model and most likely he will start from scratch again.

Although the mainstream culture in the art department is still waterfall based rather than AGILE, it is also true that most of existing tools don't support approaches based on refinements and probably these two aspects are related.

A modeling system based on polar meshes can actually support an AGILE approach,

not only because the designer can step by step refine the mesh up to the desired level of details, but also because there is an implicit skeleton that can be used to automatically rig the mesh to the bones. The designer at this point is free to adjust and refine a completely animated character without compromising the entire rigging and animations already set up.

Moreover it is also possible to select entire features, to copy them into other models or to save them for later usage. This is the base of a new generation of object libraries, where the old *copy-paste* pattern is replaced by the more flexible *recombine-customize*: the upper body of an alien and the lower body of a humanoid can be mixed together, can be refined to adjust the mesh details and a complete new character is served. Rigged, animated and textured for free.

Waterfall and AGILE development

Waterfall [Roy70] is a linear development process where the project follows a sequential life cycle: the new development phase starts only when the previous has been fully completed. This was the first method that has been applied to the development of digital products and it often leads to unbounded delays and increments of the production cost.

The antithesis of waterfall is AGILE [HF01], a term introduced in 2001 to group a class of iterative and incremental development methods, where the digital product evolves in rapid cycles: in each iteration self organized and cross functional teams execute the entire production pipeline, from requirements specification to release and consequently at the end of each cycle a working product is always available for reviews. The feedback that is possible to gain at the end of each iteration is an important component of the AGILE method and it allows the team to start the new cycle with the most updated requirements.

The benefits of the AGILE approach has been largely proven in the software development world, but the same principles can be applied straight forward in almost any digital production.

1.3.3 Animations

Some of the benefits of a mesh with an implicit skeleton is the animation system, which can be simplified to support a more direct application of the standard IK technique.

Polar meshes provide also a good support to physically based behaviors, computed in real time using the structural information intrinsically embedded in the mesh topology: the global deformation can be implemented as a combination of localized

behaviors which are substantially independent from the specific skeleton shape and that can be therefore applied to a broad range of different meshes.

A completely different world of possibilities is provided by morphing: features and skeleton segments give meaningful information about the object structure and they can be exploited to coherently blend the mesh among two states. The deep understanding of the mesh structure that the algorithm have at run-time is so powerful that it is possible to dynamically add, change or remove entire mesh features, such as character tails and horns.

1.3.4 Procedurally generated content

Another set of applications of the new modeling system is related to procedurally generated content. This vast area can clearly benefit from a mesh topology that includes clear structural information and that can be exploited to simplify the design of a broad range of algorithms, from procedural shape deformation to graphical representations of L-Systems.

Another interesting application is the automatic tweaking of the level of detail, that can be achieved in both direction exploiting the properties of polar meshes: it is possible to automatically reduce the level of detail, rating the importance of each edge compared to its position in the structure and evaluating a possible simplification, but it is also possible to analyze the curvature of the surface in order to increase its detail level (for certain extent) without compromising the shape defined by the designer. This can be done using different interpolation techniques.

1.4 The project

The main focus of this project is therefore exploring the benefits that a modeling system based on polar meshes can lead, regarding the pure object modeling as well as the animation system and the production of procedurally generated content.

Although these three aspects will be independently analyzed and presented in the three main parts of this document, it is important to acknowledge how they relate to each other and their own relative weight.

The modeling tool isn't only a good start point for the project but it is also a fundamental prerequisite for the animation system and the procedurally generated

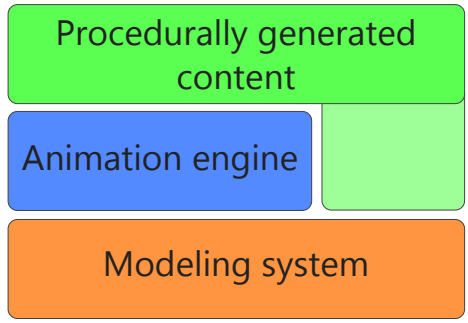


Figure 1.4: Project structure

content, and both those aspects use the same core operations defined in the modeling tool. The same role of prerequisite is also played by the animation engine towards the procedural content creation when the output of the algorithm is animation poses rather than (or additionally to) object shapes.

The core of the modeling system, will be described in the homonymous part, while the following sections will explore the possibilities that the modeling tool can offer on top of that. The reader is highly recommend to follow the document step by step, or in case she/he needs a quicker, although incomplete, overview on the project, the suggestion is to read the modeling part and the conclusions, which are both definitely a must, while the discussion regarding animation systems and procedurally generated content can be eventually skipped.

Part I

Modeling

Chapter 2

A new modeling system

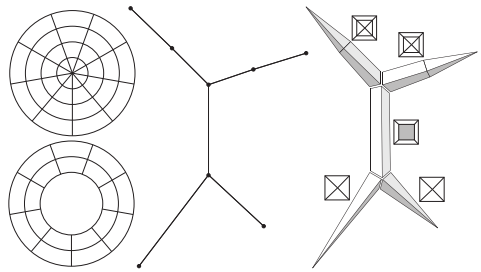


Figure 2.1: Polar mesh: examples of polar (top left) and annular (bottom left) regions. (source: [JAB12])

2.1 Concept of polar mesh

Most of the interesting models in computer graphics are, or they are composed by, subdivision surfaces[Wik] and in these cases the mesh of the object is just a numerical approximation of the smooth surface that the designer wants to represent. In this sense it is possible to obtain a better representation of the underlying surface using subdivision methods and techniques.

As presented by Thomas J. Cashman, over the past 15 years different subdivision schema have been developed, and nowadays they are a fundamental component of every modeling system. The standard de facto in this field is Catmull-Clark, the "very first subdivision schema for surfaces" [Cas12] that inspired several others, all based on recursion of simple rules.

An easy way to define subdivision surfaces is using "a sequence of nested spline rings converging to an extraordinary point" [KP07a], and in this case Catmull-Clark (and most of the other methods) poorly performs, especially when the valence is high.

Karčiauskas and Peter, instead, proposed an alternative layout where the valency can be progressively adjusted in such a way that if n is the valency of the extraordinary point, then each ring is a sequence of n segments smoothly interconnected. Meshes that shows this layout are called *polar regions* and they completely avoids the typical corners yield by the Catmull-Clark subdivision schema. As an extension, it is also possible to define *annular regions*, where the extraordinary point has been removed together with the triangle fan that it supports.

A mesh composed by an interconnected set of polar and annular regions is called *polar mesh [JAB12]*and it supports and has a closure under the polar subdivision schema.

2.2 SQM: Skeleton to Quad Mesh

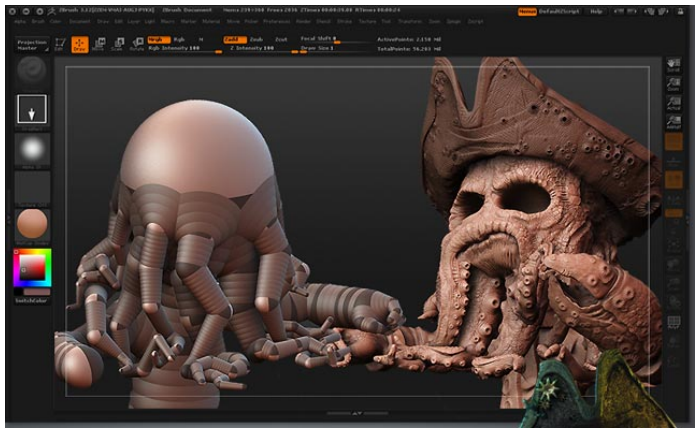


Figure 2.2: ZBrush: usage of ZSpheres to define the model's structure (left) before creating the polygonal mesh (right). Source: http://pixologic.com/

Skeletons are the de facto standard for most of the animations that are daily implemented in the 3D industry, however in the typical production pipeline the mesh is firstly created and later skinned to match its skeleton. It is important to notice that, in this case, the CGI artist doesn't exploit the shape information that he can get from an easy to create skeleton: not only the high poly mesh is created directly from scratch but later it has to be manually skinned to match its skeleton, so the designer has to deal twice with a complex mesh object rather than exploit a simple skeleton structure.

In their work J. A. Bærentzen, M. K. Misztal and K. Welnicka flipped all of this: they developed an algorithm to procedurally generate a polar mesh based on a given skeleton (SQM: Skeleton to Quad Mesh[JAB12]). A similar approach is also used in Z-Brush, the well known modeling software, and it requires that the CGI Artist designs a skeleton for the object before the mesh generation.

In SQM the CGI artist models the skeleton, that is considerably simpler than a full high poly object, then he just needs to procedurally generate the final mesh. In figure 2.3 it is shown an example.

This method surely requires CGI artists with good abstraction skills, as it isn't always easy to recognize the right skeleton for a desired polygonal mesh, but it is incredibly faster than any traditional sculpturing tool.

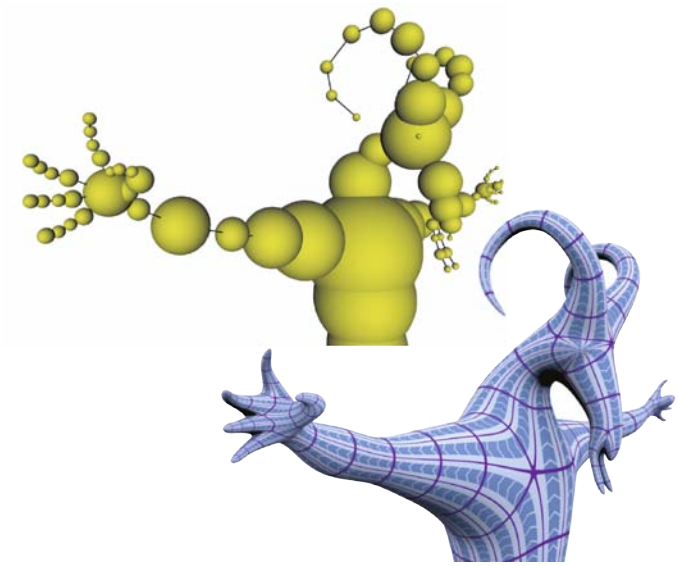


Figure 2.3: SQM: example from Bærentzen's paper [JAB12]

2.3 This project as an evolution of SQM

In his thesis [Don12], Michael Mc Donnell proved that SQM is a fairly good tool to model rough organic objects, but it suffers from severe limitations when the object that needs to be shaped is inorganic or it hasn't a clear structure. His initial assumption was that these limitations can be overcome with a richer set of skeleton nodes. This wasn't only supported by Leblanc's work [LLP11], but somehow it also makes sense: the skeleton defines the branching structure of the entire objects but it lacks of all the skin details. Imagine to find the bones of an alien, not the skin just the skeleton; is it really possible to infer its silhouette? Not really. In the same way SQM can guess the skin of the skeleton that has been modeled, but additional information is needed in order to correctly infer the right mesh.

It turned out that a richer set of nodes isn't enough to substantially expand the expressiveness of the SQM method, that remains strongly limited in its original field.

From this point of view, this thesis represents an evolution of the SQM method, but it does it from different considerations, those leads to a completely different modeling system.

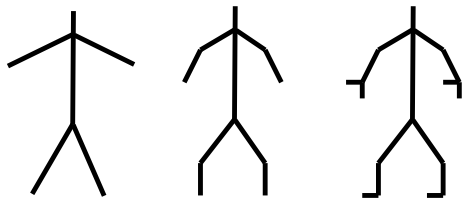


Figure 2.4: Example of different level of details for a humanoid skeleton

A comparison of the head and the gear modeled with the (improved) SQM showed both sides of the problem: the character head is pour because it hasn't a clear skeleton structure, while the gear has a correct structure but it lacks of details (the final vertices are in the wrong positions).

Together these two observations lead to a more general problem: in each mesh it is surely possible to identify a skeleton structure that can (and should) be exploited to speed up the modeling, but there are also mesh details, parts of the mesh that aren't affected by the skeleton structure but that are important to model high quality objects.

Defining what is skeleton and what is mesh detail is a pure design choice.

In figure 2.4 it is shown the skeleton of an humanoid at different levels of details. It is easy to imagine a full human body on top of each skeleton and, from this point of view, moving to a more complex bones structure reduces the features that are defined as mesh details.

The same concept can be seen comparing a traditional modeling system (Maya, 3DS Max..) where each mesh feature is in fact a mesh detail, with SQM where everything is skeleton and the concept of mesh detail has been completely removed.

> *In a good modeling system there must be a balance between mesh skeleton and mesh details.*

This will be one of the basic assumptions behind this project, and it is the result of considerations on Mc Donnell's work as well as on some of the design choices in Z-Brush.

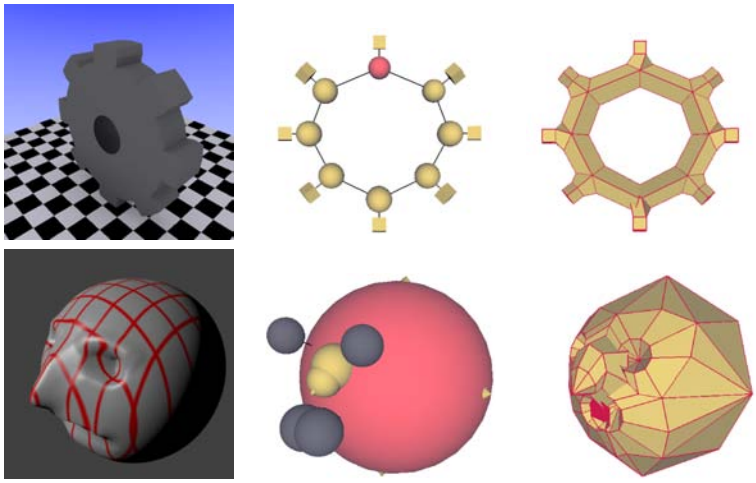


Figure 2.5: Example of SQM: Comparison of the head and the gear, images from [Don12]

2.4 The new modeling system

One of the main goals of this thesis is the definition of a modeling system than can offer a balanced combination between the needs of an high level modeling (structure definition) and low level modeling (mesh details).

Two different set of operations will be used to make the definition of the mesh structure (and consequently its skeleton) easy and fast, as well as to model mesh details with precise vertices placement:

The High level modeling: a set of operations that interact with the polar structure of the model:

- **Add/Remove Refinement**: increase or decrease the valency of a pole
- **Add/Remove feature**: create or remove a branch on the specified position.
- **Merge/Split feature(s)**: merge two polar regions on a single annular region. In the inverse operation an annular/polar region is split in two separate polar regions.

Low level modeling: the mesh details instead can be fully defined with the 3 most traditional mesh operations. Please note how none of the following tools can affect the mesh structure in any way:

- **Move**: translate the selected group of vertices
- **Scale**: increase or decrease the relative distance of a group of vertices
- **Rotate**: spin the selected group of vertices around its relative center

The benefits of this approach will be proved developing a prototype that will strictly support only this new modeling paradigm, where the mesh operations will be closed towards the polar/annular property (each operation will be applied to a polar mesh and it will output a polar mesh) and the mesh skeleton will be inferred run-time directly from the mesh vertices. In order to test the real expressiveness of this approach no additional post-processing will be supported.

2.5 Consideration on mesh topology

2.5.1 Triangles vs Quad dominant meshes

Although the rendering systems historically support mesh based on arbitrary (convex) polygons, today the entire modeling world processes only quad or triangles based meshes. The main reason for this evolution is the huge performance benefits leaded both by the limited number of vertices per primitive and the hardware parallelization that characterizes modern GPUs; but although both the recent GPUs and

the rendering engines are designed, implemented and optimized for triangle based meshes, designer and CGI Artists often prefer quads.

Quad dominant meshes have two big benefits [Dil] compared to triangles based meshes:

Subdivision methods: there are subdivision methods that work on triangles as well ($\sqrt{3}$ Subdivision by Kobbelt [Kob00] for example), but quad based subdivision methods give, in general, a better control to the designer regarding the desired level of detail and they yield more balanced meshes.

Edge loops: another interesting aspect is the definition and preservation of edge loops, set of connected edges that define close rings. It is easy to prove that general triangle meshes don't easily yield edge loops, simply due to their odd number of vertices per primitive, but edge loops are an important feature to avoid artifacts on animations that involve blending and folding.

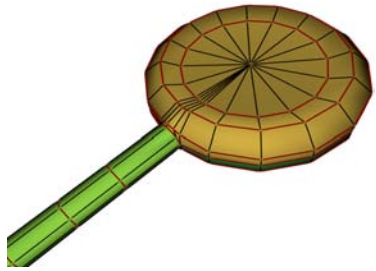


Figure 2.6: Polar Mesh: edge loops highlithed in red.

2.5.2 Polar meshes from a topology prospective

Polar meshes are surely an interesting case of quad dominant meshes and they fully preserve edge loop structure: as previously defined a polar mesh is a smoothly interconnected set of polar and annular regions, each of them defined as a sequence of spline rings. It is trivial to see that each spline ring is in fact an edges loop.

Polar meshes naturally supports the polar subdivision schema [KP07b], a subdivision methods that changes the valency of the polar/annular region without compromising the loops of edges.

Chapter 3

Polar Meshes

Polar meshes can be more formally defined with few definitions:

Annular region If $Q = (q_1, q_2..q_n)$ is a tuple of quads, each of them defined by four edges $q_i = (h_{i1}, h_{i2}, h_{i3}, h_{i4})$, Q is called an annular region if $\forall q_i \in Q \quad h_{i1} = h_{(i-1)3} \wedge h_{i3} = h_{(i+1)1}$. In this case the tuples $L_1 = (h_{12}, h_{22}, ..h_{n2})$ and $L_2 = (h_{14}, h_{24}, ..h_{n4})$ are nested spline rings and they are called loops of the annular region.

Pole and pole region A pole is a vertex p center of the triangle fan $T = (t_1, t_2..t_n)$, marked as a "pole". If each triangle is defined by $\forall t_i \in T \quad t_i = (p, h_i, h_{mod_n(i+1)})$, than $L = (h_0, ..h_n)$ is called loop of the pole. T, the triangle fan itself is called pole region.

Loop A loop is a tuple of edges $L = (h_0, ..h_n)$, where each edge $h_i \in L$ is defined by two vertices $h_i = (v_i, v_{mod_n(i+1)})$.

For each **loop** in the mesh, there are **at least** 2 elements that share the loop, which are either annular or polar regions;

For each **loop edge** in the mesh there are **exactly** 2 elements that share the loop, which are either annular or polar regions;

Backbone A backbone is a connected path of non-loop edges either between two poles or that loop itself:

- if the backbone, defined as the tuple of edges $B = (h_0, ..h_n)$, is a path between the poles p_1, p_2, then each edge is defined by two vertices $\forall h_i \in B \setminus \{h_0, h_n\} \quad h_i = (v_i, v_{i+1})$, $h_0 = (p_1, v_2)$ and $h_n = (h_{n-1}, p_2)$.
- if the backbone, defined as the tuple of edges $B = (h_0, ..h_n)$, is a closed path, then each edge is defined by two vertices $\forall h_i \in B \quad h_i = (v_i, v_{mod_n(i+1)})$.

In both cases none of the internal edges can be part of a loop: $\forall h_i \in B \quad \neg\exists L$ in the mesh $| \, h_i \in L$ (design constrain).

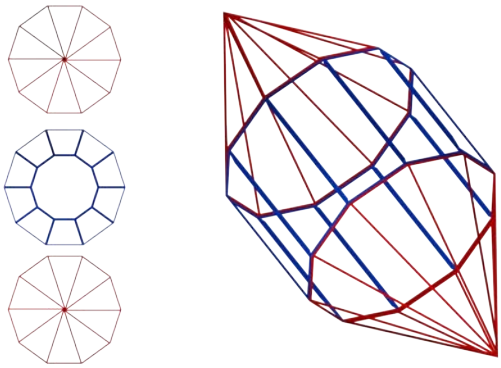


Figure 3.1: Polar mesh: example

3.1 Notation

In order to keep a precise but concise description, for the following sections a mathematical notation will be used to describe polar mesh components and some basic operations that affect them:

3.1.1 Mesh elements

V	set of vertices of the mesh
$P \subseteq V$	set of poles of the mesh
H	set of halfedges of the mesh
F	set of faces of the mesh

3.1.2 Properties of mesh elements

3.1.2.1 Vertices

$v = vert(h)$	vertex $v \in V$ pointed by the halfedge $h \in H$
$V_x = vert(H_x)$	$V_x = \{vert(h) \mid h \in H_x \subseteq H\}$
$H_x = out(v)$	set of halfedges H_x outgoing from the vertex v
$H_x = out(V_x)$	$H_x = \{out(v) \mid v \in V_x \subseteq V\}$
$p = pos(v)$	position of the vertex v in global coordinates

3.1.2.2 Faces

$f = face(h)$	face $f \in F$ associated with the half edge $h \in H$.
$F_x = face(H_x)$	$F_x = \{face(h) \mid h \in H_x \subseteq H\}$
$H_x = halfedges(f)$	$H_x = \{h \in H \mid f = face(h)\}$, in other words H_x is the set of halfedges that defines the face f
$H_x = halfedges(F_x)$	$H_x = \{halfedges(f) \mid f \in F_x \subseteq F\}$

3.1.3 Basic mesh navigation

$h_2 = opp(h_1)$	if $h_1 \in H$ exists between $v_1 \in V$ and $v_2 \in V$, such as $v_1 = vert(h_1)$, then $h_2 \in H$ exists between v_1 and v_2 and $v_2 = vert(h_2)$
$h_2 = next(h_1)$	h_2 immediately follows $h_1$in the definition of $f = face(h_1)$, with $h_1 \in H$ and $h_2 \in halfedges(face(h_1))$
$h_2 = prev(h_1)$	if $h_1 = next(h_2)$, with $h_1 \in H$ and $h_2 \in halfedges(face(h_1))$. In other words h_2 immediately precedes $h_1$in the definition of $f = face(h_1)$

3.1.4 Basic mesh operations

$v = split\_edge(h)$	splits the half edge $h \in H$ in half by creating the new vertex $v \in V$.
$split\_face(f, v_1, v_2)$	splits the face $f \in F$ by creating a new edge between v_1 and v_2, with $v_1, v_2 \in vert(halfedges(f))$
$merge\_edges(h_1, h_2)$	if $p_x = vert(h_1) = vert(opp(h_2))$ then it merges the two halfedges, removing the vertex p_x.
$merge\_faces(h, f_1, f_2)$	merges the faces f_1 and f_2 removing the common edge h
$remove\_vertex(v)$	removes the vertex v from V
$remove\_edge(h)$	removes the halfedge h from H
$remove\_face(f)$	removes the face f from F
$(h_1, h_2) = create\_edge(v_1, v_2)$	creates a pair of halfedges between the vertices v_1 and v_2, such as $h_1 = opp(h_2)$, $vert(h_1) = v_1$ and $vert(h_2) = v_2$
$f = create\_face(V_x \subseteq V)$	creates a new face f from the list of vertices V_x

3.2 Topological Properties

Some of the topological properties that characterize polar meshes will be highlighted in this section. These properties are important to understand the reasons of some of the design choices that will be later presented in this report.

3.2.1 Poles arbitrariness

Prop It isn't always possible to uniquely identify the poles from the mesh topology.

Proof In figure 3.2 it is shown a regular octahedron. The symmetry in the mesh makes the choice of the poles (in red) completely arbitrary and both of the proposed solutions are possible.

Consequences As poles can't be dynamically inferred on the fly, it is necessary to explicitly store the list of vertices that assume this role in the mesh. This set of pole markers can be created when a new mesh is imported into the system and it has to be updated after the execution of each mesh operation.

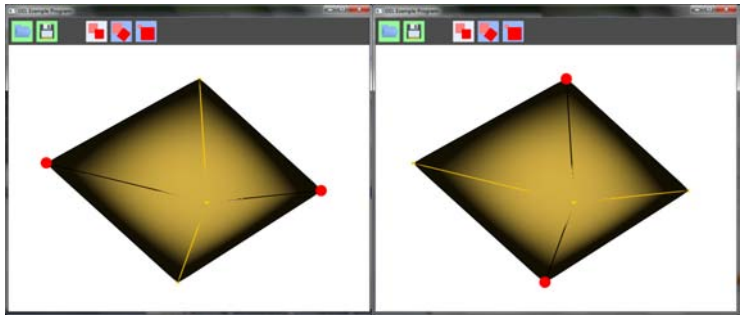


Figure 3.2: Poles arbitrariness: visual proof

3.2.2 Loops arbitrariness

Prop It isn't always possible to uniquely identify the loops from the mesh topology.

Proof In figure 3.3 it is shown an example where the choice of loops and backbones is completely arbitrary.

Consequences As loops can't be dynamically inferred on the fly, it is necessary to explicitly store the list of halfedges that assume this role in the mesh. This set of markers can be managed in the same way that the list of poles has been handled: it can be created when a new mesh is imported into the system and it has to be updated after the execution of each mesh operation.

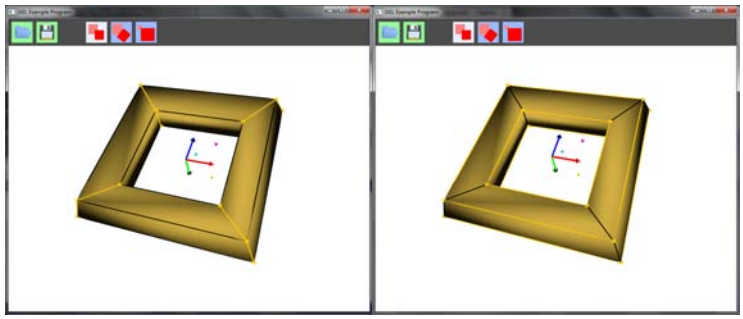


Figure 3.3: Loops arbitrariness: visual proof

3.2.3 Loop and backbone connectivity

Prop

- If h is a loop edge, then $next(h)$ is a backbone edge.
- If k is a backbone edge and $vert(k)$ isn't a pole, then $next(k)$ is a loop edge.

Proof To demonstrate both sentences, lets analyze annular and polar regions separately:

- In annular regions each quad is by definition a sequence of backbone-loop-backbone-loop edges, therefore both propositions are trivially true.
- In polar regions instead each triangle is defined by a loop edge followed by 2 backbone edges. As the pole is always between these last two, it is easy to see that both statements are true also in polar regions.

3.3 Iterators

A key prerequisite of all the operations that have been implemented is their ability to navigate the mesh. Some halfedge iterators has been built for this purpose:

Loop iterator: given an half edge, it walks along the entire loop or backbone where the half edge lies.

Parallel iterator: given an half edge, it iterates through all the half edges that are parallel to the initial one.

Fan iterator: given a pole, it iterates through all the halfedges outgoing from the pole

3.3.1 Loop iterator

Input	$h_0 \in H$
Output	$O = \{h_0\} \cup \{next(opp(next(h)))\|h \in O \wedge vert(h) \notin P\} \cup \{prev(opp(prev(h)))\|h \in O \wedge vert(opp(h)) \notin P\}$
Function	$O = Loop(h_0)$

3.3.2 Parallel iterator

Input	$h_0 \in H$
Output	$B = Loop(next(h_0))$ if $vert(h_0) \notin P$ $B = Loop(prev(h_0))$ otherwise. $O = \{next(h) \vert h \in B \wedge vert(h) \notin P\}$
Function	$O = Parallel(h_0)$

3.3.3 Fan iterator

Input	$p \in P$
Output	$O = \{h \vert h \in H \wedge vert(opp(h)) = p\}$
Function	$O = Fan(p)$

3.4 Features

3.4.1 Definitions

Nesting Loop A nesting Loop is a Loop of the mesh that supports a breach in the mesh skeleton. More formally a nesting loop is a loop $L = (h_0, ..h_n)$ where $\exists h_i \in L \vert valency(h_i) > 4 \wedge vert(h_i)$ isn't pole.

Skeleton segment A skeleton segment is what most of 3D modeling systems call *bone* and it is an abstract representation of an annular or polar region:

Annular region: defined by the loops L_0 and L_1 implicitly defines the skeleton segment S between the points $P_0 = \frac{1}{|L_0|} \sum pos(vert(L_0))$ and $P_1 = \frac{1}{|L_1|} \sum pos(vert(L_1))$.

Polar region: defined by the pole P and the loop L implicitly defines the skeleton segment S between the points P and $P_1 = \frac{1}{|L|} \sum pos(vert(L))$.

Feature A feature is a group of polar and annular regions that implicitly defines a chain of consecutive and connected skeleton segments between 2 nesting loops, or 2 poles or 1 nesting loop and 1 pole. These elements are called boundaries of the feature.

3.4.2 Child features

The skeleton of a polar mesh is an undirected graph therefore there is no notion of parent-child relation between features. It is important to acknowledge that SQM yields tree based skeletons instead, and due to some of its design choices, it does have a parent-child relation between skeleton nodes, while it doesn't natively support loops. Both these aspects will have an important impact on the capabilities of the newly designed modeling system.

Proof In figure 3.4 it is shown a clear example where it is impossible to infer any parent-child relationships between features. In facts all these scenarios are possible:

- The feature associated with pole A is a child of the features associated with B-C
- The feature associated with pole B is a child of the features associated with A-C
- The feature associated with pole C is a child of the features associated with A-B

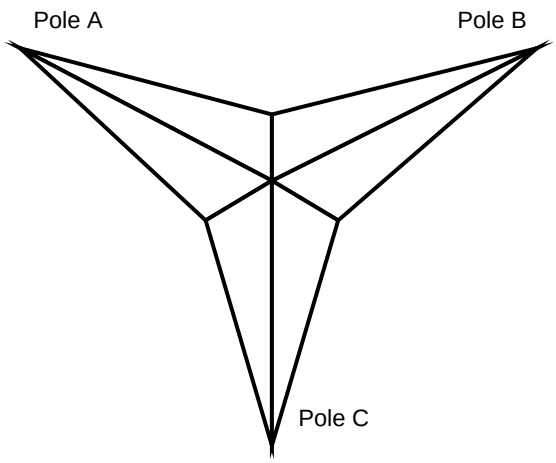


Figure 3.4: Example polar mesh where it is impossible to infer a parent-child relationship between features

Chapter 4

Core Operations

In this chapter it will be presented the 7 polar mesh operations that have been designed, implemented and tested. These operations represent the core of the high level modeling system and they concretely increase the abstraction level of the tool that has been developed.

4.1 Add refinement

Add a loop or a backbone at the specified point. Conceptually:

- when a loop is added, it generates a new annular region from an existing polar or annular region
- when a backbone is added, it increase the valency of the interested regions.

4.1.1 Specification

```
1 | void addRefinement(halfEdge h0)
```

h0 is one of the half edges those will be splitted during the operation. In general $H_0 = Parallel(h_0 \in H)$ are all and only the affected half edges.

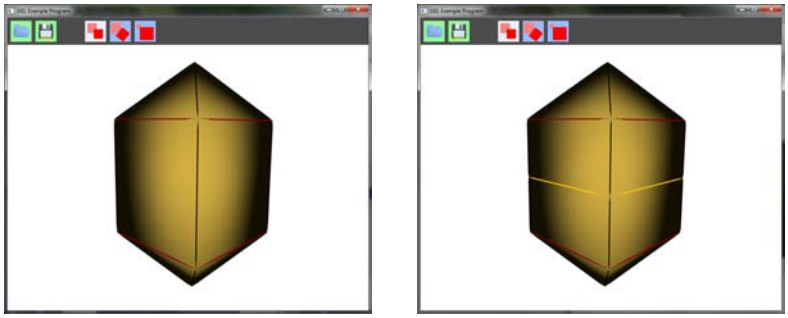


Figure 4.1: Example of *addRefinement* used to add a loop

4.1.2 Implementation

```
1  list toSplit
2  for h in H0:
3      p = split_edge(h)
4      definePosition(p)
5      toSplit.push_back(h,p)
6
7  for i in [0..toSplit.length]:
8      h = toSplit[i].h
9      p0 = toSplit[i].p
10     p1 = toSplit[(i+1)%toSplit.length].p
11     if(vert(next(h)) is Pole):
12         p1 = vert(next(h))
13     split_face(face(h),p0,p1)
14
15     newH = halfedge(p0,p1)
16     if(!isLoop(h0)) setLoop(newH)
17     else setLoop(next(newH))
18
19     px = vert(prev(opp(h)))
20     if(px is Pole):
21         split_face(face(opp(h)),p0,px)
22         newH = halfedge(p0,px)
23         setLoop(next(newH))
```

4.1.3 Description

Add refinement is an operation to add loops and backbones to the mesh:

- to add a loop, addRefinement needs to be applied to one of the backbone edges that is going to be splitted with the new loop.
- to add a backbone, addRefinement needs to be applied to one of the loop edges that is going to be splitted with the new backbone.

The algorithm operates in 3 steps:

1. it collects all the edges that are parallel to the given parameter, until it closes a loop or it will reach the extreme poles (standard usage of the parallel iterator).
2. it splits the collected edges with new vertices
3. it splits the faces associated with the collected edges, generating the new refinement and it marks the newly created edges as loops and backbones.

4.1.4 Limitations

AddRefinement can be applied to every edge of the mesh without any specific limitation.

4.2 Remove refinement

Remove a loop or a backbone from the mesh. Conceptually:

- when a loop is removed, it merges an annular region into a polar region or into another annular region
- when a backbone is removed, it reduced the valency of the interested regions

4.2.1 Specification

```
1 | void removeRefinement(halfEdge h0)
```

h0 is one of the half edges that are part of the refinement that needs to be removed. In general all edges in $H_0 = Loop(h_0 \in H)$ will be removed at the end of the operation, which can be applied if and only if $\forall h \in Loop(h_0) \quad Loop(h) =$

$Loop(h_0)$, or in other words if and only if the selected half edge isn't part of a loop where a feature is nested.

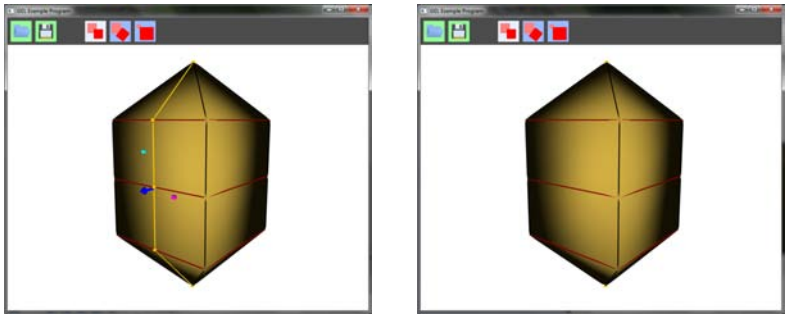


Figure 4.2: Example of *removeRefinement* used to remove a backbone

4.2.2 Implementation

```
1 list toRemove
2 for h in Loop(h0):
3     f1 = face(h)
4     f2 = face(opp(h))
5     if(vert(h) is not Pole):
6         toRemove.push_back(opp(next(h)))
7     merge_faces(h,f1,f2)
8 for h in toRemove:
9     merge_edges(h,next(h))
```

4.2.3 Description

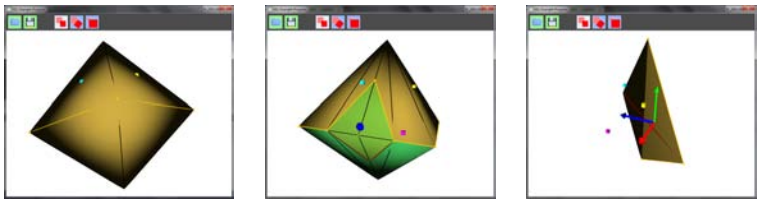
This operation can be used to remove a loop or a backbone from the mesh. In both cases it must be applied to an edge of the loop or backbone you wish to remove. *removeRefinement* applies two steps:

1. It removes each edge in the refinement, merging the two adjacent faces
2. It removes the unnecessary vertices

4.2.4 Limitations

In order to keep the structure of the mesh coherent, this operation can't be applied:

- To remove the last loop that keeps two poles separated (Figure 4.3-a).
- To remove loops that support nested features (Figure 4.3-b). In order to remove these refinements it is necessary to remove all the nested features first.
- To decrease to 0 the valency of one or more regions (Figure 4.3-c).



(a) (b) (c)

Figure 4.3: *removeFeature*: examples where it isn't possible to proceed.

4.3 Add Feature

It adds a new feature and it nests it at the specified position. The size of the gap that will host the new feature and its valency are defined with the number of input vertices.

4.3.1 Specification

```
1 | void addFeature(vertex[] VV)
```

VV collection of vertices that defines the size of the supporting gap and the valency of the new feature.

- Constrains:
  - $\forall p \in VV \quad p \in V \wedge p \notin P$
  - VV is a collection of consecutive vertices on the same loop, therefore $\nexists h \in H | vert(h) \in VV \wedge vert(next(h)) \notin VV \wedge vert(next(next(h)) \in VV$
  - If p is the newly created pole, $|out(p)| = 2 * (|VV| - 1)$

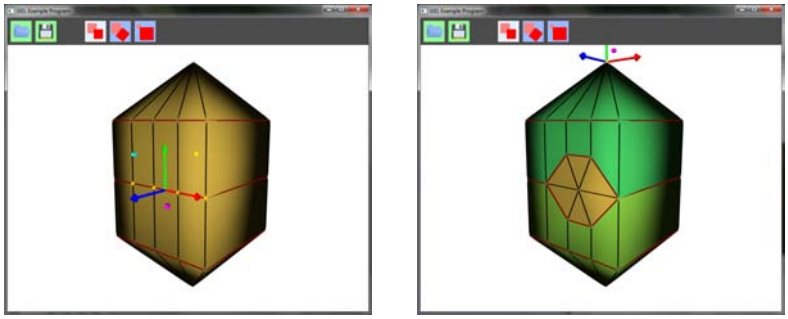


Figure 4.4: Example of *addFeature*

4.3.2 Implementation

```
1  HalfEdge extremeLeft, extremeRight
2  extremeLeft = getLeftMostVertex(vv)
3  extremeRight = getRightMostVertex(vv)
4  loop = getInterestedLoop(vv)
5
6  H = Loop(opp(extremeLeft))
7  list newVerts
8  newVerts.push_back(vert(extremeLeft))
9  for h in H:
10     if(h==extremeRight): break
11     newTop = split_edge(h,h+1)
12     split_face(face(h),newVerts.back(),newTop)
13     setLoop(halfedge(newVerts.back(),newTop))
14     newVerts.push_back(newTop)
15
16     definePosition(h)
17     definePosition(newTop)
18     if(h!=extremeLeft) merge_faces(face(h),h-1)
19
20 split_face(face(h),newVerts.back(),extremeRight)
21 setLoop(halfedge(newVerts.back(),extremeRight))
22 merge_faces(face(h),h-1)
23
24 Vertex p = split_face_vertex(face(h))
25 setPole(p)
```

4.3.3 Description

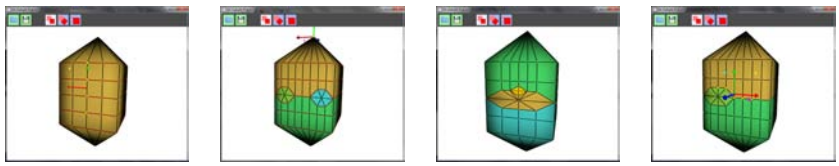
The algorithm follows 4 steps:

1. It identifies the left most and right most vertices of the input

2. It iterates over the selected vertices left to right, splitting the top faces in order to generate the upper side of the loop gap. It reuse the existing loop to define the lower side and it sets the newly generated edges as loop components.
3. It merges the inner faces in a single big polygon
4. It creates the final triangle fan splitting the big polygon with a vertex

4.3.4 Limitations

- Features must be nested on skeleton nodes, therefore it isn't possible to define features along backbones, only on loops (Figure 4.5-a).
- It is possible to nest several features on the same skeleton node, therefore it is possible to define multiple features on the same loop (Figure 4.5-b).
- It is possible to define features on loop edges that are the boundary of an existing nested feature (Figure 4.5-c).
- It isn't possible to define features only partially overlapped with other features or, in other words, if a loop supports nested features, then it must support the entire features' gap (Figure 4.5-d).



(a) (b) (c) (d)

Figure 4.5: *addFeature*: examples.

4.4 Select Feature

Identify and select all the components of the feature associated with the given face.

4.4.1 Specification

```
1 | set<faces> selectFeature(face f)
```

f face of the mesh

return the set of faces that are part of the same feature of f

4.4.2 Implementation

```
ret = new set<faces>()
b = getBackboneEdge(f)
B = Loop(b)
featureFound=false
for backbone in B:
    if(featureFound): break
    if(isPole(vert(backbone))):
        fanLoop = Fan(vert(backbone)):
        for l in fanLoop:
            ret.push_back(face(l))
        break
    loop = Loop(next(backbone))
    for l in loop:
        ret.push_back(face(l))
        if(valency(vert(l))>4):
            featureFound=true

if(featureFound):
    B = Loop(opp(b))
    for backbone in B:
        if(featureFound): break
        if(isPole(vert(backbone))):
            fanLoop = Fan(vert(backbone)):
            for l in fanLoop:
                ret.push_back(face(l))
            break
        loop = Loop(next(backbone))
        for l in loop:
            ret.push_back(face(l))
            if(valency(vert(l))>4):
                featureFound=true

return ret
```

4.4.3 Description

First of all, the algorithm identifies a backbone edge of the input face, then it moves forward and backwards along that backbone adding each loop to the result set. The process stop when one of the following conditions is met:

- The backbone loops and the algorithm is at the original start point.

- A pole is reached and the polar region is added to the result set.
- A vertex with higher valency is detected, unique sign of the presence of nested features.

4.5 Remove Feature

Remove the feature associated with the given face.

4.5.1 Specification

```
1 | void removeFeature(face f)
```

f face associated to the feature to remove.

- Constrains:
  - In the loop that represent the boundary of the feature there must be exactly two vertices with valency higher than 4 (split vertices)
  - Along the boundary loop, the split vertices are connected with two different paths. These paths must have the same number of internal vertices.

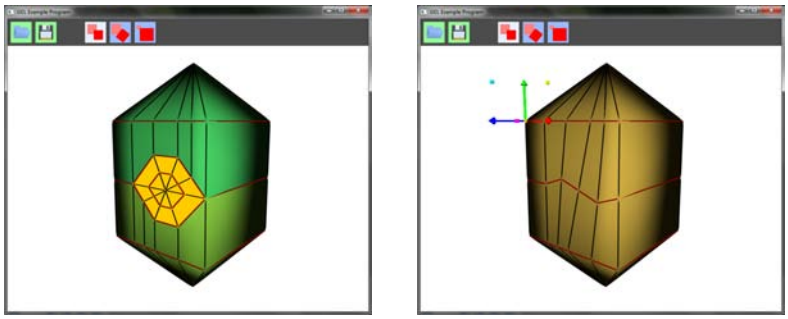


Figure 4.6: Example of *removeFeature*

4.5.2 Implementation

```
pole = getFeaturePole(f)
B = Loop(out(pole))
// 1 - Structural check
featureFound = false
for backbone in B:
    if(featureFound): break
    loop = Loop(next(backbone))
    for l in loop:
        if isSplit(l):
            if(!isGoodSplitPoint(l)) return
            splitEdge = l
            stopLoop = next(backbone)
            featureFound=true
            break

// 2 - Remove internal loops
featureFound=false
while(!featureFound):
    HalfEdgeID b = out(pole)
    HalfEdgeID h = next(b)
    L = Loop(h)
    for li in L:
        if(li.isSplit()): featureFound=true
    if(!featureFound) removeLoops(h)

// 3 - Prepare edge lists
L1 = Loop(splitEdge)
L2 = Loop(splitEdge)
list faceTop, faceBottom, edgeTop, edgeBottom
i1=1
i2=0
do:
    i2=(i2+1)%L2.length
    i1 = (i1-1)%L1.length
    l1=L1[i1]
    l2=L2[i2]
    faceTop.push_back(face(ck.opp(l1)))
    faceBottom.push_back(face(opp(l2)))
    edgesTop.push_back(l1)
    edgesBottom.push_back(l2)
    pos(vert(l2)) = (pos(vert(l2))+pos(vert(opp(l1))))/2.0f
while(vert(l2)!=vert(opp(l1)))

// 4 - Remove triangle fan
F = FanIterator(pole)
for fi in F:
    remove_face(face(fi))
    remove_edge(opp(fi))
    remove_edge(fi)

remove_vertex(pole)

// 5 - Fix Structure
eTop = edgesTop.begin()
eBottom = edgesBottom.begin()
```

```
56 fTop = faceTop.begin()
57 fBottom = faceBottom.begin()
58 while(eTop!=edgesTop.end()):
59     set_vert(opp(eTop),vert(eBottom))
60     set_vert(opp(next(opp(eTop))),vert(eBottom))
61
62     set_vert(eTop,vert(opp(eBottom)))
63     set_vert(prev(opp(eTop)),vert(opp(eBottom)))
64
65     set_face(eBottom,face(opp(eTop)))
66     connect_next_and_prev(prev(opp(eTop)), eBottom,,next(opp(eTop)))
67
68     set_last(fTop,eBottom)
69     set_out(vert(eBottom),opp(eBottom))
70     set_out(vert(opp(eBottom)),eBottom)
71
72     if(vert(eTop)!=vert(opp(eBottom)))
73         remove_vertex(ck.vert(eTop))
74
75     remove_edge(opp(eTop))
76     remove_edge(eTop)
77
78     ++eTop
79     ++eBottom
80     ++fTop
81     ++fBottom
```

4.5.3 Description

To remove an existing feature several steps are necessary:

1. Check the structure to identify the split vertices (valency higher than 4). If more than 2 split vertices are detected or if they are placed in inconvenient positions, the operation is aborted.
2. Remove all internal loops to reduce the feature to a triangle fan
3. Adjust vertices position
4. Remove the triangle fan
5. Close the gap, paring and merging top and bottom faces.

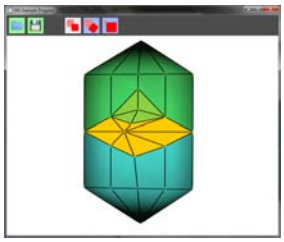
4.5.4 Limitations

A first important limitation is actually related to the current prototype implementation: it is possible to remove features only if they are bounded between a loop and

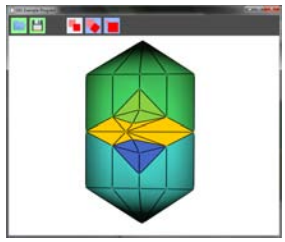
a pole. Features bounded by two loops can't be removed straight away but they need to be firstly splitted and then removed with a distinct operation.

This algorithm is also affected by other limitations, more deeply connected to the modeling system: in facts in its last step, it can actually compromise the polar structure of the mesh unless some conservative constrains are maintained. In order to safely remove the feature there must be exactly 2 split vertices on the boundary loop of the mesh to remove (figure 4.7-b) and they need to be connected with two different paths, which must have the same number of internal vertices (figure 4.8).

It is easy to demonstrate that these constrains are sufficient but not necessary (in figure 4.7-a for example it is theoretically possible to remove the yellow feature but it isn't allowed in the current prototype), nonetheless it is important to maintain the polar structure and, on the other hand, none of these conditions really limits the expressiveness of the system.

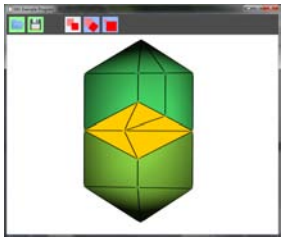


(a) operation possible, but unsupported

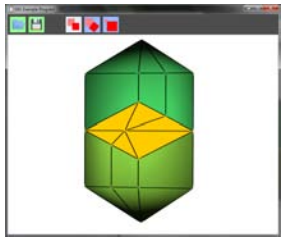


(b) operation impossible

Figure 4.7: *removeFeature*: limitations due to nested features.



(a) operation impossible



(b) operation possible

Figure 4.8: *removeFeature*: symmetric limitations.

4.6 Merge Features

Merge two features, bridging the poles

4.6.1 Specification

```
1 void mergeFeature(vertex p1, vertex p2)
```

p1 pole associated with the first feature

- Constrains: $p1 \in P$

p2 pole associated with the second feature

- Constrains: $p2 \in P$, $valency(p2) = valency(p1)$

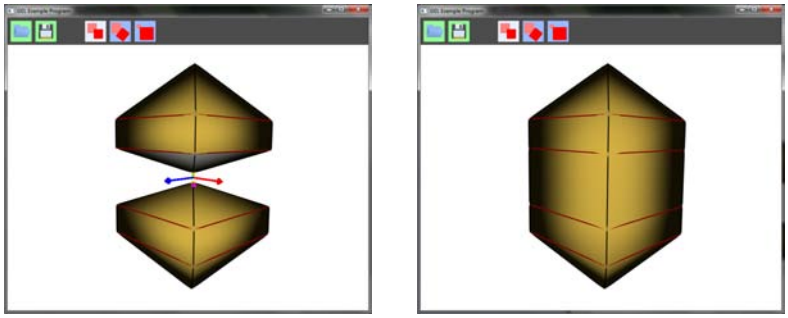


Figure 4.9: Example of *mergeFeature*

4.6.2 Implementation

```
1  fan1 = Fan(p1)
2  fan2 = Fan(p2)
3
4  // 1 - find index of vertecis at min distance
5  (i1,i2)=vertecis_min_distance(fan1,fan2)
6
7  // 2 - remove triangle fans
8  for i in [0..fan1.length]:
9      remove_face(face(fan1[i]))
10     remove_face(face(fan2[i]))
11 arrayList edges
```

```
12
13 // 3 - bridge loops
14 for i in [0..fan1.length]:
15     pair(h1,h2)=create_edge(vert(fan1[i]),vert(fan2[i]))
16     edges.push_back(pair(h1,h2))
17 for i in [0..fan1.length]:
18     hh0 = fan1[i]
19     hh1 = edges[i].h1
20     hh2 = opp(fan2[i])
21     hh3 = edges[(i-1)%fan1.length].h2
22     create_face(hh0,hh1,hh2,hh3)
```

4.6.3 Description

In order to merge two features, identified by the 2 given poles, the following steps are necessary:

1. Find a match between the vertices of the two base loops. In the current implementation the risk of a twist effects has been minimized using the closest vertices to initialize the pairing procedure.
2. pairing the closest vertices together.
3. Remove the two triangles fans
4. Bridge the loops, generating new halfedges and new faces between the matched vertices.

4.6.4 Limitations

mergeFeature can be applied to any pole pairs, as long as their valencies matches.

4.7 Split Feature

Split the feature in two parts, replacing a loop with two poles

4.7.1 Specification

```
1 void splitFeature(halfedge h0)
```

h0 halfedge part of the loop that needs to be replaced by the two new poles

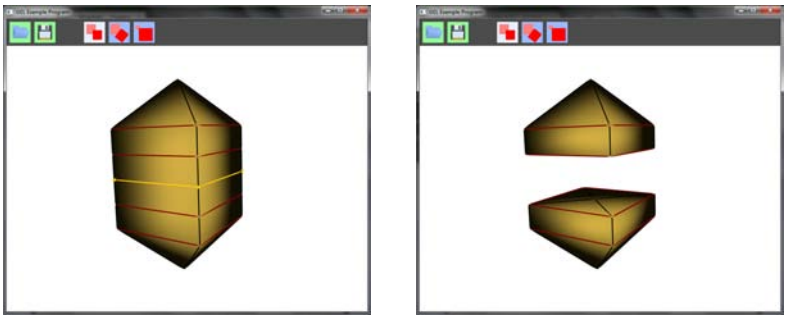


Figure 4.10: Example of *splitFeature*

4.7.2 Implementation

```
1  split = Loop(h0)
2  before = Loop(opp(next(next(h0))))
3  after = Loop(next(next(opp(h0))))
4
5  // 1 - remove current bridge
6  for h in split:
7      remove_face(face(opp(h)))
8      remove_face(face(h))
9
10 // 2 - cove holes with single polygon
11 f1 = create_face(verts(before))
12 f2 = create_face(verts(after))
13
14 // 3 - create poles in the middle of new faces
15 newPole1 = split_face_by_vertex(f1);
16 setPole(newPole1)
17 newPole2 = split_face_by_vertex(f2);
18 setPole(newPole2)
```

4.7.3 Description

In order to split two features, it is important to have 3 loops next to each other: a loop that can support the first new triangle fan (feature 1), the loop that needs to be replaced by a two poles (split loop) and finally a loop that can support the second new triangle fan (feature 2).

This sandwich structure is important in order to correctly split the mesh. Additionally no nested feature are allowed on the split loop, which will be removed.

Under these conditions the split operation requires:

1. To remove the split loop and the ring of faces on both its sides.
2. Generate 2 polygons to cover the holes on both sides, along the two supporting loops
3. Split the newly generated polygons with a vertex in order to convert them into triangle fans. Both new vertices will be marked as poles.

The triangle fans can be generated in different ways of course, but this is surely one of the easiest.

4.8 Example

In this section the modeling process of the a gear has been described step by step. This is an illustrative example than can give a deeper understanding on how the different polar mesh operations interact with the model as well as how low level sculpturing tool have been integrated into the system.

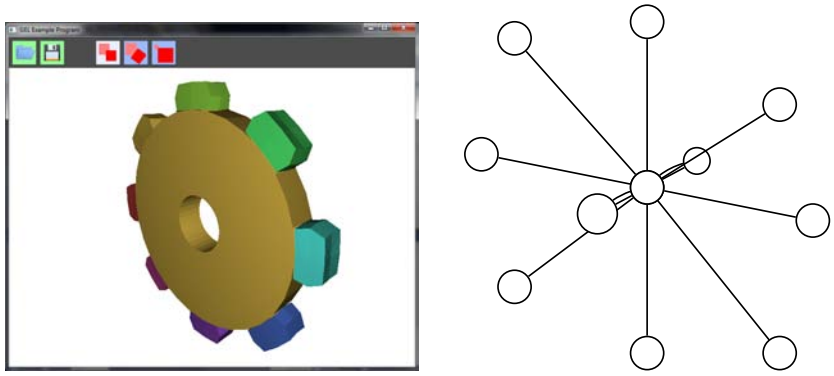
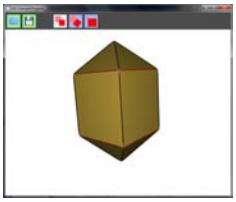


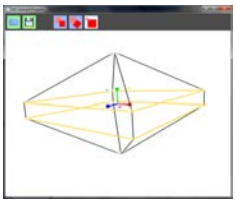
Figure 4.11: Gear: goal of the following modeling process (mesh and skeleton)

1 A procedurally generated regular octahedron has been used as starting point for the modeling process.

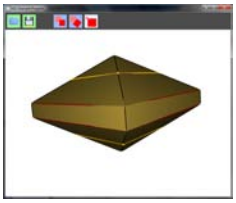
2 Two refinement operations have been used to add the highlighted loops.



3 The middle loop has been removed with *removeRefinement*

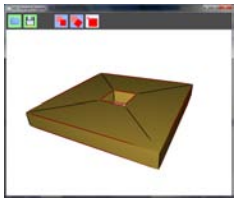


4 The highlighted loops have been scaled and repositioned. They will be the external edges of the gear disk. (Low level modeling)

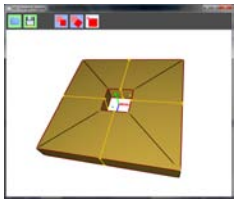


5 Other two loops have been added with *addRefimenent*. These will be the internal edges of the gear disk, defining the central hole.

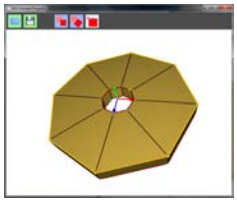
6 Like in (4), the newly added loop have been scaled and repositioned. (Low level modeling)



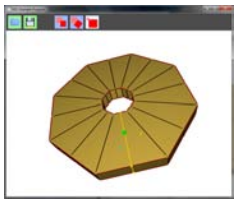
7 The two poles have been merged together, replacing the corresponding polar regions with an annular region (the faces that define the central hole)



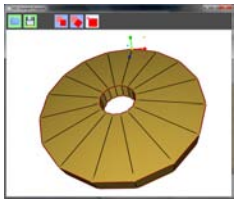
8 With *addRefinement*, 4 more backbones have been added to the model, doubling the valency of all the annular regions.



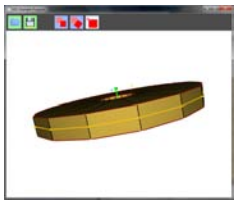
9 The new vertices have been repositioned to shape the model as a raw disk (Low level modeling)



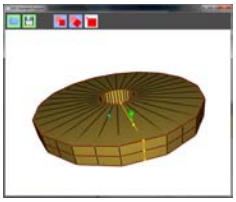
10 Like in (8), *addRefinement* has been used to add 8 more backbones



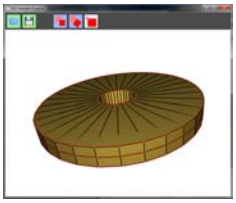
11 Again, the new vertices have been repositioned to shape the model as a disk (Low level modeling)



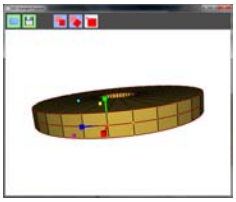
12 addRefimenent has been used one more time to split the external annular region. This middle loop will support the features that represent the gear teeth.



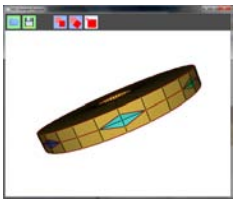
13 As the disk hasn't the desired level of detail yet, a new series of addRefinement operations double the valency of the annular regions again.



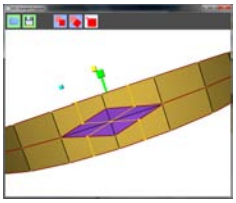
14 Consequently a new adjustment of the vertices positions is necessary. (Low level modeling)



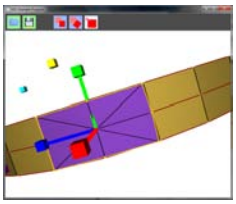
15 At this point the mesh is ready to support the new features



16 Multiple application of *addFeature* have been used to create the teeth of the gear. In the following steps the demonstration will focus on a single tooth only.

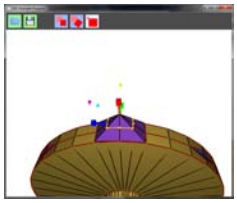


17 The valency of the newly created pole has been increased with *addRefinement*.

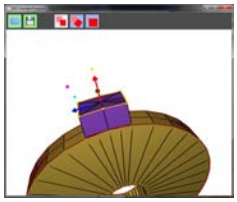


18 The vertices at the base of the feature have been repositioned to have a squared gear tooth. (Low level modeling)

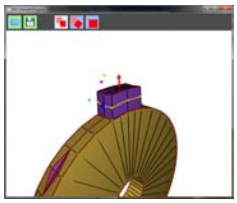
19 The pole has been moved upward to extrude the tooth. (Low level modeling)



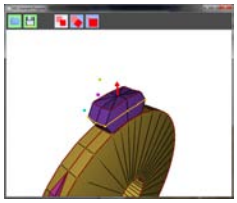
20 *addRefinement* has been used one more time to create the upper edge of the tooth.



21 The newly created loop has been scaled and moved according to the desired shape. (Low level modeling)



22 Another refinement operation has been applied to add a middle loop to the feature.



23 In the final step the position of the vertices have been adjusted to reach the final hexagonal shaped tooth. (Low level modeling)

Chapter 5

Analysis

It is important to test the expressiveness of the modeling system in order to have a better understanding of the limits that a design tool based on polar meshes involves. An important set of tests have been performed to compare the power of the current modeling system with the SQM algorithm [JAB12] which is in many ways the father of this project. According to Michael Mc Donnell [Don12], SQM suffers from some important limitations especially regarding non organic objects. He investigated the limits of the SQM approach defining some interesting modeling goals and evaluating the performances of the SQM algorithm towards them. He also proposed some extensions to that modeling system in order to improve those performances.

It is interesting at this point to evaluate the behavior of the proposed modeling system against the same modeling goals, to visualize the improvements that have been reached:

- Modeling of a Lollipop
- Modeling of a Gear
- Modeling of a Ladder
- Modeling of a Head

Each object will be presented in its desired shape (goal), modeled by SQM, modeled with the improved SQM from Mc Donnell and finally modeled using the current prototype. All screenshot are part of Mc Donnell's work [Don12], expect the ones related to the current implementation, of course.

5.1 Lollipop

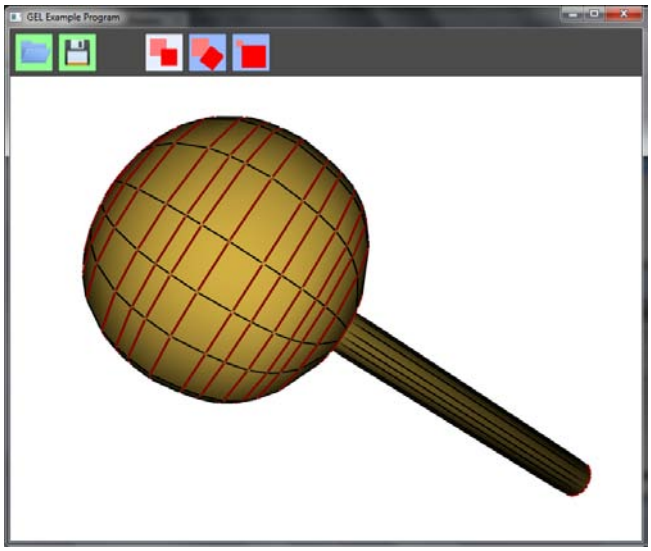
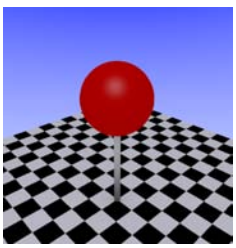
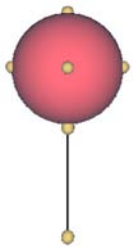


Figure 5.1: Test result: Lollipop

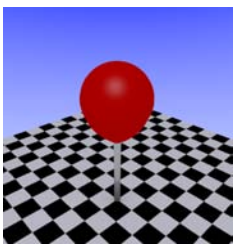
Figure 5.2: Lollipop: skeleton of the new model



Goal



Skeleton (SQM)



SQM

The model produced with SQM is already very close to the goal, but there is still room for improvements: the shape of the lollipop is more a drop than a sphere and the base of the stick isn't perfectly flat [Don12]. These problems aren't related in anyway with the structure of the object and they are difficult to solve with skeleton based operations (SQM). The new modeling system overcame easily both of them thanks to low level sculpturing tools that let the designer to define the precise position of each individual vertex.

Another problem that affects the model produced in SQM is related to the skeleton structure: in order to use it in future animations, it is important that the skeleton reflects the structure of the object. Although SQM doesn't provide a method to procedurally generate a final skeleton for the model, it is interesting to notice how the structure that has been used to produce the mesh, includes quite unnaturally, 4 branches to shape the lollipop sphere. The new model, instead, doesn't use any branch to define the object (in figure 5.1, the mesh is painted in a single color, therefore it has only one feature) and the entire structure is reduced to a sequence of skeleton segments, as expected. In this case it is actually possible to use the procedurally inferred skeleton straight forward in the animation engine.

5.2 Gear

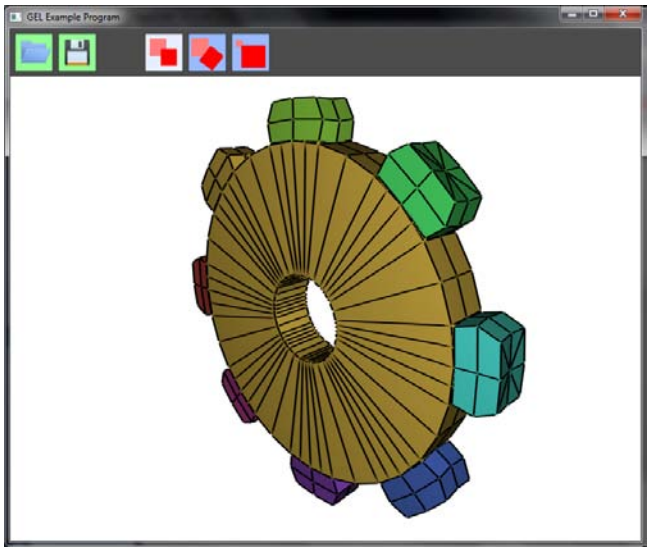
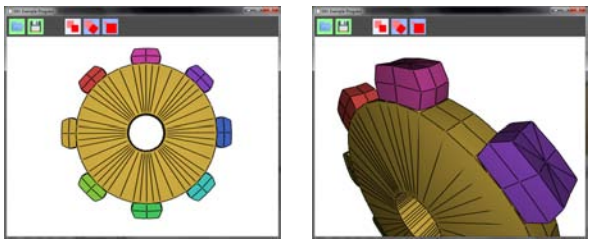
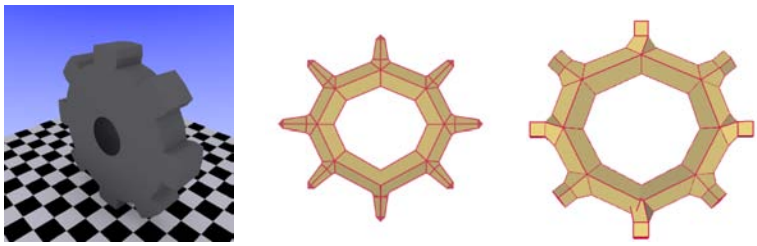


Figure 5.3: Test result: Gear



(a) Front view (b) Close view

Figure 5.4: Test result: Gear, front, close view and skeleton



Goal SQM Improved SQM

A gear is a good example of inorganic object as it is characterized by sharp edges and large flat surfaces. The model produced with SQM is a very pour representation of this object, that failed not only in details like the rotation of the gear teeth, but also in its basic structure that has been modeled as a pipe, rather than a flat disk. This is a general problem in SQM, where the size of the section of a skeleton segment can be influenced but it isn't practically possible to shape it: although Mc Donnell worked on this aspect providing different types of skeleton nodes, SQM remains very limited compared to a tool that allow free vertices placement.

Great improvements have been made with the new modeling system, that produces an high quality mesh much closer to the goal and that includes some tiny details like the hexagonal section of the gear teeth. At a closer look, the new mesh still shows some imperfections, small vertices mispositions that can be easily fixed improving the low level operations of the prototype, for example including the standard sculpturing tools already available in most commercial software (first of all, global and local coordinate system to move, scale and rotate vertices).

It is also worth to mention that the new model took 10-15 minutes to be completely developed, an amount of effort that can be partitioned according to the 80/20 law: the first 20% of the time has been spent with high level modeling operations, to define the basic structure of the object, while the remaining 80% has been

spent adjusting the individual vertices (low level modeling). SQM is clearly a faster development method but, on the other hand, it limits itself to an high level modeling tool and, from this point of view, the effort required to produce a complete object in SQM is comparable to the effort spent with high level structural definition in this new system.

5.3 Ladder

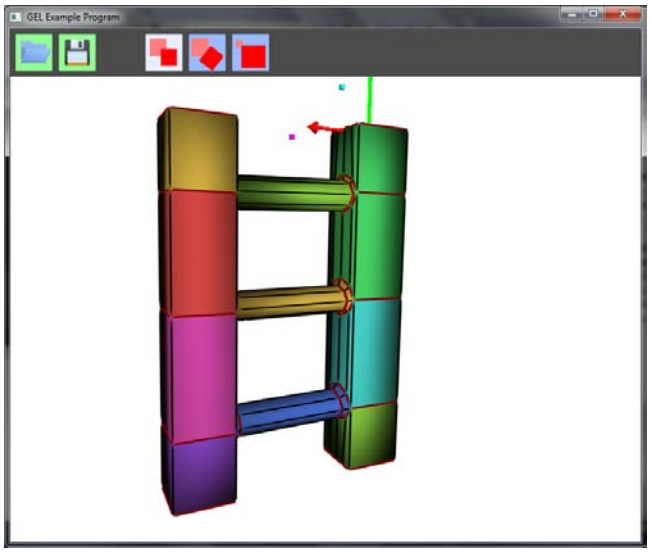
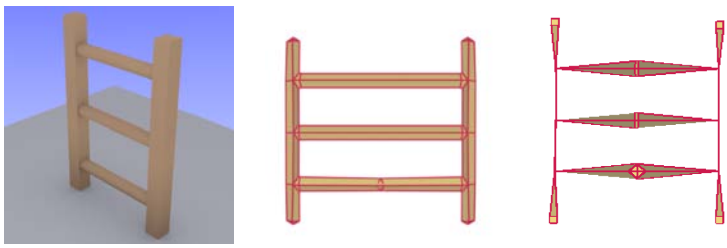


Figure 5.5: Test result: Ladder



Goal SQM Improved SQM

The ladder is another example of inorganic object, that the new modeling system is able to produce almost perfectly.

In this case it is interesting to notice how the basic SQM method performs better than the improved SQM, which uses different node types to model details like the shape of the rungs section or a flat end in the ladder structure. The reason of this behavior is related, one more time, on how mesh details are implemented: in the new system they are handled with simple vertices displacements, which can't affect the skeleton structure, but in SQM they have to be part of the skeleton and this clearly interferes with its anatomy.

Another advantage of the new modeling system is the level of detail, which can be locally controlled in each individual part of the mesh: the ladder produced with SQM has been kept extremely low poly in order to preserve the sharp edges on the main structure, inducing equally raw rungs as well; while the new model is able to differentiate these areas and, to some extent, it shows fairly round rungs in a low poly support.

5.4 Head

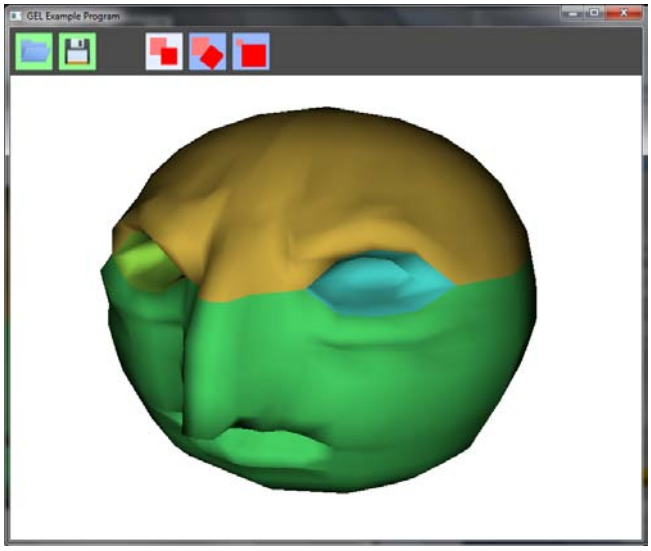
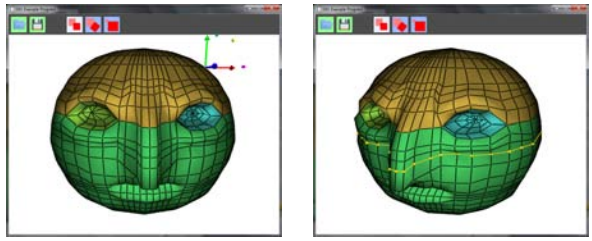
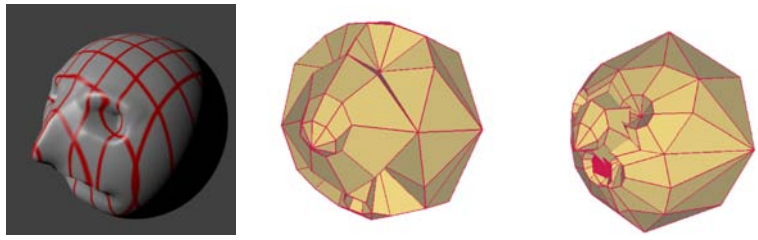


Figure 5.6: Test result: Head



(a) (b)

Figure 5.7: Test result: Head, alternative views



Goal SQM Improved SQM

Although SQM seems a good approach for most organic objects, it poorly performed in this task and the technical limitations that Mc Donnell well documented in his report (lack of effective support for concavities, issues related to quads alignment..) are only part of the problem: in this case the features that need to be modeled, in order to produce the head, are almost entirely mesh details, simple vertices displacements of a sphere that SQM has to force into artificial skeleton nodes to be able to model.

The new system instead, embraces the idea that some of the characteristics of the object aren't related to its structure and that can be more conveniently defined with traditional mesh tools. The result isn't perfect yet, mainly due to some technical limitation of the current implementation, which make low level modeling tricky and time consuming, but on the other hand, it is fairly easy to theoretically prove that the head can be actually drawn with any desired level of detail: the target object can be in facts horizontally scanned with an arbitrary number of sections and each of them will be a different loop in the model. The valency of these annular regions can also be arbitrary chosen, defining the number of vertices per loop, whose positions can be freely adjusted according to the correspondent scan of the target object, creating a deformable lattice dense as needed.

This test highlights an important aspect of the modeling of organic objects, first of

all characters: SQM can be used to quickly produce a basic mesh but in practical cases there are a lot of details that are simply impossible to model with skeleton bones and a completely different approach is necessary. In this perspective, an integration between a modeling system based on polar meshes and a traditional sculpturing tool can be an effective solution.

5.5 Conclusions

In all the performed tests, the new modeling system produced fairly convincing results and it was able to overcome the limitations that deeply affected the SQM method.

The new models contains a richer set of mesh details that they reproduce with more fidelity from the desired target object. This has been reached integrating traditional sculpturing tools (low level modeling) in the polar modeling system, allowing the designer to adjust the position of each individual vertex.

This is a feature that SQM isn't able to offer, not only because its implementation lacks of vertices manipulation tools but also because it is a one way process: in SQM there isn't a bidirectional map between the skeleton and the mesh, therefore each adjustment in the structure of the model completely overwrites the entire model (Figure 5.8). In the new system instead, the modeling process have been pushed towards a more AGILE approach where, exploiting the structural information naturally embedded in the polar topology, both high and low level operations have been specifically designed as non disruptive mesh improvements (Figure 5.9).

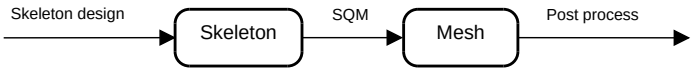


Figure 5.8: Modeling process of SQM

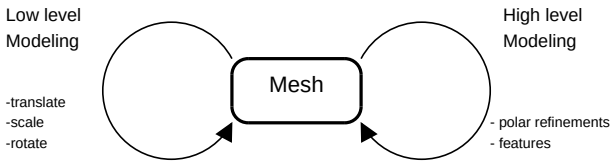


Figure 5.9: Modeling process of the new modeling system

Finally, another important improvement that have been reached is related to the skeleton, which now reflects more naturally the intrinsic structure of the object: SQM uses the skeleton anatomy to include mesh details in the model, but this

generates objects with artificial and overcomplicated bones structures that are difficult to model and that can't be used for animations without a dramatic (manual) simplification.

The new modeling system moreover, allows any combination of polar and annular regions, providing full native supports to skeletons based on non-simple graphs. This overcomes some limitations of SQM that requires tree based skeletons and forces a branch on the root node, therefore models with loops and holes, like the ladder, can now be shaped in their real structure rather than faked in a post process.

Although in general the new modeling system is a positive proof of concept, it also leaves room for some improvements: the low level modeling system needs to be expanded with the set of tools that already characterize most of commercial 3d software, while symmetric constrains and copy-paste of features can optimize the high modeling operations.

Part II

Animation

Chapter 6

Animation engines

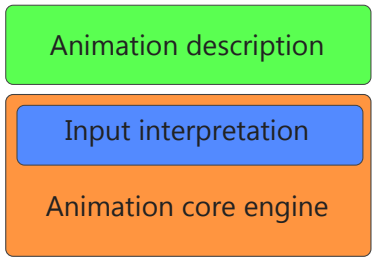


Figure 6.1: Animation system: Structure

Animations are a fundamental part of all modern 3D productions, from videogames to movies and simulations. During the years, several animation systems have been developed , based on different principles and approaches, but they all are grounded by a common property: they have been developed independently from the modeling system and they don't make any assumption regarding the mesh structure.

This is a clear advantage in term of applicability, but on the other hand it limits their possibilities in terms of expressiveness: every animation system is in facts an algorithm to procedurally move the mesh vertices according to specific high level

inputs and, from this point of view, animation engines use basic modeling operations as a low level API library.

It makes sense, at this point, to investigate if a different modeling system can lead to an improved animation engine and if more advanced modeling features can be exploited to implement complex dynamic behaviors.

Conceptually an animation system can be split in two distinct layers: a language to describe the desired mesh deformation and an internal component that actually implements those behaviors (figure 6.1).

6.1 Animation description

Animations can be described in several different languages, each of them characterized by its own level of details. The most common methods [ana] to describe complex deformations are:

- **Keyframes**: an old and simple technique, originally developed in the Walt Disney studios [BW, Las87], where the designer defines the poses (vertices position) in correspondence of specific time stamps, called keyframe, that the system linearly interpolates in order to approximate the full configuration set in all the in-between frames.
- **Animation scripting languages**, where the deformation is defined using descriptive languages. A typical example of animation scripting is the Improv System [PG96], developed by Ken Perlin and Athomas Goldberg at the Media Research Laboratory (New York University).
- **Goal directed motions**, where the animator specify the desired final behavior and the animation system computes the motion to reach or to performs that behavior.
- **Motion capture**: a physical movement is tracked, recorded and coded into a virtual animation. In this case the deformation isn't designed in a virtual space but it is initially performed by an actor, a person or an object, who actually moves in the real world, while a specific system keeps track of the movements with dedicated hardware and software.

For the readers who aren't familiar with them, a more detailed description of these techniques can be found in the appendix, including a short discussion on the pros and cons of each method.

6.2 Animation engines

The internal animation engine is where the input is processed and the movements are performed. The principle itself is very simple: providing an updated position for the mesh vertices at every screen refresh.

As most of animations aren't described with complete per-frame configuration sets, interpolation is a fundamental component of every animation engine: in all key frame based animations, including motion capture, interpolation is obviously necessary in order to fill up the in-betweens, to uniquely define the full configuration set for every non-key frame; but interpolation still plays an important role in goal oriented motions and scripted animations, not only because there are cases where the dynamic description of the configuration set is incomplete in some of the timestamps, but also because the simulation engine that applies physical forces, performs the dynamic behavior and/or executes the scripts can be independent from the pure animation engine and it can work on a different rate. For example in Unity3D the physical simulation is executed at a fixed interval (controlled by the developer), while the animation engine still needs to produce a configuration set per frame (and the frame rate isn't controllable by the developer) [unib].

Whatever it has been used to define the motion, key framing rather than a more dynamic description, interpolation isn't a trivial task: in the general case a pure linear interpolation performs poorly and more complex techniques should be used to convey smoothness and naturalism in the animation. This has been clearly described by John Lasseter in his paper "Principles of traditional animation applied to 3D computer animation" [Las87], but it is also present in the modern game industry that Martin Jonasson and Petri Purho well represent with a famous talk on *juicing* [jui].

Chapter 7

Animation of polar meshes

Obviously the basic structure of a traditional animation system can be applied also to polar meshes, but an animation system specifically developed for this topology can exploit some of its properties to gain benefits that a more general animation system can't offer.

The most important aspect of polar meshes is the possibility to procedurally infer the mesh skeleton from the model with a clear and well structured skinning. This leads to an animation system that can describe complex animations as a combination of basic local behaviors, that affect singular skeleton nodes without an extensive knowledge of the entire skeleton structure.

7.1 Inferring the skeleton

The mesh skeleton can be inferred with a fairly simple recursive algorithm, that performs a depth-first graph visit of the mesh. Variants of this algorithm are also used in the *selectFeature* operation and in the rendering module to highlight different features in different colors.

Although these variants have specific per-node operations, the basic recursive struc-

ture is a constant:

SkeletonInit explores the current feature, loop after loop, following a specific backbone. This function relies on *skeletonInitLoop* and *skeletonInitPoleLoop* to explore respectively annular and polar regions. Here new skeleton nodes are created and linked to each other.

```
1  skeletonInit(HalfEdge backboneEdge, Node parent):
2     Backbone = Loop(backboneEdge)
3     bi = arg(Backbone, backboneEdge) //set i such as Backbone[i
          ]=backboneEdge
4     i = bi
5     featureFound=false
6
7     //boundary loop for nested feature
8     if(isPole[vert(opp(b))]==0):
9        for li in Loop(prev(b)):
10          processedVertex[vert(li)]=true
11          centers[vert(li)]=parent
12          offsets[vert(li)]=pos(vert(li))-parent.targetPosition
13
14    for i in [bi .. Backbone.lenght]:
15       if(featureFound): break
16       Node n = new Node()
17       parent->addChild(n)
18       if(isPole(vert(Backbone[i]))):
19          skeletonInitPoleLoop(vert(Backbone[i]),n)
20          break
21
22       h = next(Backbone[i])
23       featureFound = skeletonInitLoop(h,n)
24       parent = n
```

SkeletonInitLoop explores a single mesh loop to pair each vertex to its skeleton node (skinning). Whenever a nested feature is detected, it is queued and later processed with *skeletonInit*.

```
1  bool skeletonInitLoop(HalfEdge loopItem, Node current):
2     H = Loop(loopItem)
3     featureFound=false
4     list nextFeatures
5     for li in H:
6        processedVertex[vert(li)]=true
7        if(li.isSplit()):
8           featureFound=true;
9           refBackbone=next(li)
10          cur=next(opp(refBackbone))
11          cur=next(opp(cur))
12          while(cur!=refBackbone):
13             nextFeatures.push_back(cur)
14             cur=next(opp(cur))
15             cur=next(opp(cur))
16
```

```
17      current.init(center(H))
18      for li in H:
19          centers[vert(li)]=current
20          offsets[vert(li)]=pos(vert(li))-center(H)
21
22      for f in nextFeatures:
23          if(processedVertex[vert(f)]==false):
24              skeletonInit(f, current)
25
26      return featureFound
```

skeletonInitPoleLoop explores the polar region and map its pole to the correct skeleton node.

```
1   skeletonInitPoleLoop(VertexID pole, Node current):
2       current.isPole=true
3       current.init(pos(pole))
4       processedVertex[pole]=true
5       centers[pole]=current
6       offsets[pole]=Vec3f(0)
```

7.1.1 Relative root node

Although the skeleton itself is an undirected graph, that is devoid of parent-child relationship, the extracted skeleton has been reduced to a rooted tree in order to simplify future skeleton visits: as the user has full control on where to place the root node, the lack of information regarding loops becomes an actual benefit in terms of simplicity and it assures that each skeleton segment will be visited exactly once.

This behavior is a design choice of the current prototype and it can be easily changed to generate the skeleton with its full graph structure, if needed.

7.1.2 Examples

In figure 7.1 it is shown how this algorithm explores the mesh, feature after feature, with the typical behavior of a depth-first graph visit. Please note how the algorithm fully supports both meshes that lead to tree-based skeletons as well as meshes that lead to graph-based skeletons (figure 7.2).

Another interesting example is shown in figure 7.3 where the algorithm explores a mesh with annular regions only.

Figure 7.1: Skeleton recognition: tree

Figure 7.2: Skeleton recognition: mesh with loops (ladder)

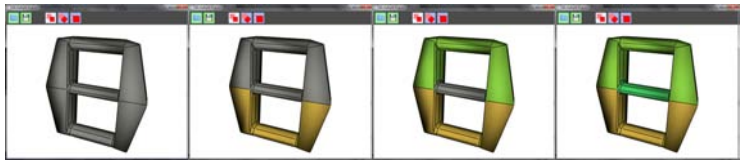


Figure 7.3: Skeleton recognition: mesh without poles

7.2 Skeleton based coordinates

An interesting aspect of this approach is the possibility to define mesh deformations in a skeleton oriented coordinate system. The position of each vertex is defined as the sum of:

- The position of its skeleton node, relative to the parent node.
- The offset of the specific vertex, relative to the position of its node.

In this way it is possible to define deformations that independently affect the structure (the position of the skeleton nodes) and the mesh details (offset of the vertices relative to their nodes).

7.3 Skinning of polar meshes

In skeletal animations, skinning is the process of assigning each vertex to one or more skeleton segments, in order to make the mesh following the skeleton when it will be animated. An important part of the animation quality is defined by the skinning algorithm which has to assure that the mesh folders properly in any possible skeleton pose.

Skinning is a complex activity and, at the state of the art, it has been only partially automated: several algorithms for automatic skinning have been proposed so far (for example [JT05] from the Computer Graphic Lab of the University of California - San Diego) but manual adjustments are often necessary to reach the desired quality level and to avoid undesired effects like the "collapsing elbow" [LCF00].

As skinning is a localized behavior, each vertex is always affected by skeleton segments in its limited neighborhood. Polar meshes provide a good support to this

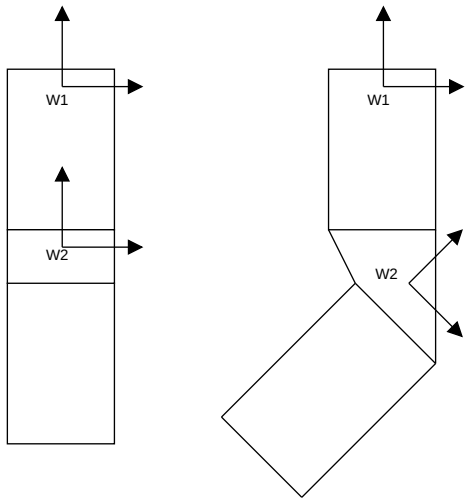


Figure 7.4: Simple Skinning method: example

activity and skinning algorithms can easily gather the structural information that is needed to weight each vertex to the proper skeleton nodes.

The approach that has been used in this thesis is what is called "simple skinning":

> "Every vertex in the mesh is attached to exactly one joint in the skeleton, and when the skeleton is posed, the vertices are transformed by their joint's world space matrix."[ski]

This basic technique assures a continuous mesh skin and, although it is clearly limited and in general it can't provide high quality results, it is fairly adequate for low poly meshes. Simple skinning can represent a good "quick and dirty" solution for the current prototype, but it is obviously inadequate for any further development and it should be replaced with a more advanced technique.

A concrete possible improvement is the "smooth skinning algorithm" [ski], where more skeleton joints contribute to define the position of each vertex, according to their weight. A specific application of this approach can use, for example, the relative angle between two consecutive skeleton segments to adjust the positions of the vertices mapped to the node between them, as shown in figure 7.5. Despite its relative simplicity, this solution would lead to considerable advantages when the skeleton is deformed with very wide angles.

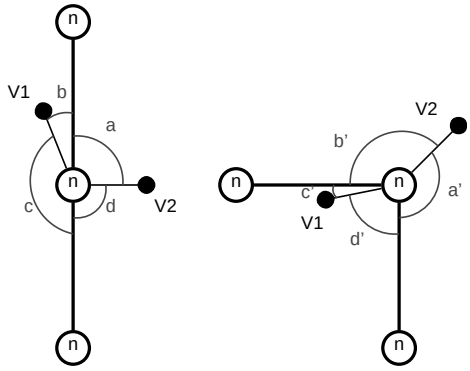


Figure 7.5: Smooth skinning: possible application with polar meshes, where the relative angle between the skeleton segments influences the position of the vertices associated with the middle node.

Chapter 8

Implementation

In the prototype a small animation system has been implemented, in order to demonstrate the applicability of some of the techniques that have been described as well as to investigate the advantages of the usage of polar meshes regarding the animation development. This isn't a complete engine, and it isn't intended to be used in real productions, but it will show some of the possibilities offered by the new modeling system.

8.1 The prototype

From a more technical prospective it has been implemented a behavior based animation system, where the final deformation is defined as a combination of a variable number of simpler behaviors.

Although these behaviors have been hard coded in the current prototype, the structure of the engine is already oriented towards a plug-in based animation system and it will be fairly easy in the future to add a script engine to the system. In this way the designer will be able to dynamically define new behaviors via scripts or graphical tools.

Despite the technique used to store and describe them, each behavior defines the evolution of a single skeleton node and the associated mesh vertices in skeleton based coordinates.

The deformation is therefore applied to the skeleton, automatically inferred from the mesh structure with the standard algorithm, and for each node it is possible to define:

Position: in the global coordinate system

Thickness: the scale factor applied to the relative distance between the vertices and their node. If this parameter is 0 then the ring of vertices collapses on the node position, while if this is 1 then all the vertices of the ring are at their original relative positions.

Root node: a reference to the node of the skeleton defined as root for the current animation.

Parent: a reference to the parent node relative to the root node

Children: reference to the children of the current node, relative to the root node

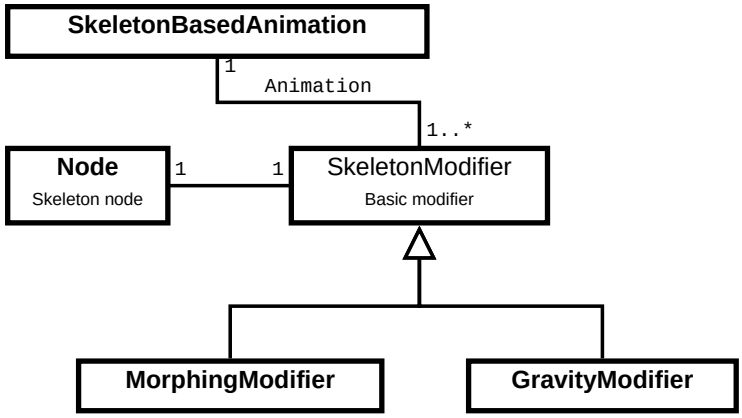


Figure 8.1: Animation system: structure of the prototype

In this framework, behaviors are coded into skeleton modifiers and they are applied independently to each skeleton node. This locality is a key feature to develop animations for completely general meshes, without any previous knowledge on their skeleton structure.

Each skeleton node can be deformed by a list of different modifiers, that are sequentially applied to define its final position: at each iteration, a general modifier in the list receives in input the position of the current node, deformed by all the previous behaviors, and it outputs its updated position to the modifier that follows.

Finally the complete animation is the global combination of these local simple behaviors.

Chapter 9

Analysis

In order to demonstrate the opportunities offered by an animation system completely tailored on polar meshes, few examples has been developed:

Morphing: the animation simulates the growing process of plants, making the mesh growing from the root node.

Gravity deforms the mesh in order to simulate the effect of the gravity. In this process the root node is considered fixed, while the rest of the mesh bends under the force.

9.1 Morphing

Morphing, an animation that changes the shape of the object, is typically unsupported by traditional skeleton based engines and in general it requires a completely different animation system. A particularly interesting example of morphing is plant development, the animation that simulates the biological growing process of weeds and trees, where morphing techniques are often mixed with traditional skeleton based transformations.

According to Callum, Przemyslaw and Campbell [GPD], there are two different approaches to simulate plant development:

Bottom-Up methods, where the final animation is the result of the evolution of single plant components: the algorithm is designed to model the deformation of each singular part and, like in L-Systems [Lin04], the global evolution is the result of a local application of growing rules. The main drawbacks of this approach are the difficulties to design local behaviors that lead to the desired final result.

Top-Down methods, where the initial and final plant shapes are set and the animation itself is merely an interpolation between these two states. This approach extends the concept of key-framing into a goal oriented animation and it is a more direct method when the focus is getting the desired shape rather than exploring the dynamics behind plant development.

In this thesis it has been used a goal-oriented top down approach with a customized interpolation system, where for each plant component it has been defined:

- The **initial state** and the **final state** of the interested plant component
- A **delay** before the interpolation is applied
- The **growth time**, the speed of the interpolation
- An **interpolation function**, a mathematical description of the interpolation itself.

The morphing behavior that has been developed in the prototype is inspired by the method presented by Callum, Przemyslaw and Campbell and it uses skeleton nodes as plant component:

- The **initial state** is the plant collapsed to it is root, while the **final state** is the plant as it has been originally modeled.
- The **state** of each node includes both its relative length to the parent node and the *thickness* of the node itself.
- The **delay** and the **growth speed** of both the length and the thickness of the node can be tweaked independently.
- It has been experimented both with a *linear* and a *sigmoidal* **interpolation**.

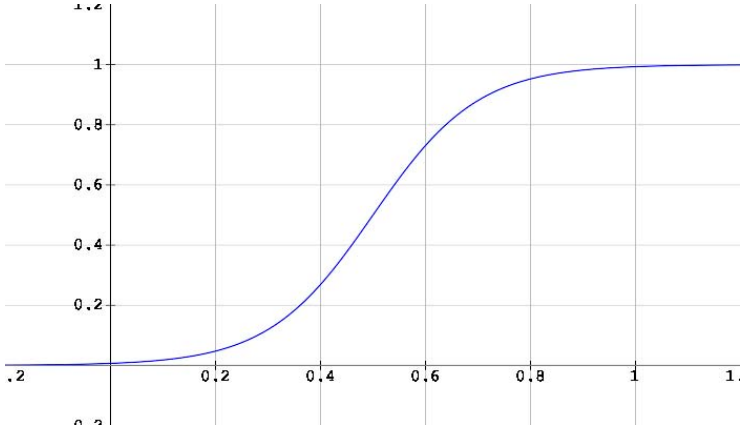
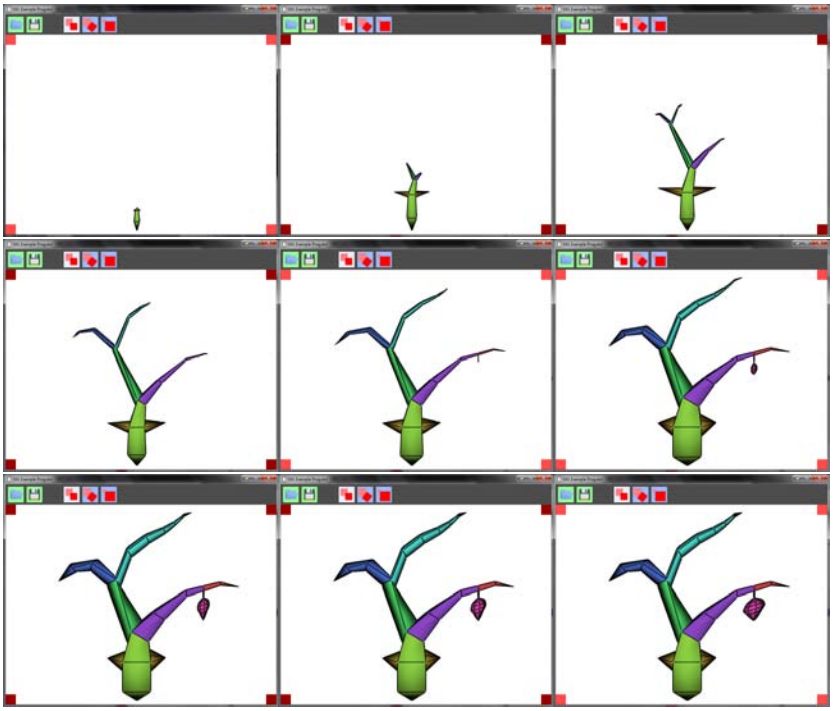


Figure 9.1: Sigmoidal interpolation used in the prototype: $f(x) = \frac{1}{1+e^{5-x*10}}$

None of the features has been removed or added during the morphing, but they are simply scaled down when unnecessary. This isn't only a design choice to simplify the algorithm itself, but it is also biologically correct as many plants develop the entire structure at the very beginning, while their "parts mature in sequence from base to apex" [Bel].

In figure 9.2, the morphing algorithm, with a basic linear interpolation, has been applied to a low poly plant. Despite the simplicity of the interpolation function, it already produces a fairly convincing plant evolution, thanks to the distribution of the loops along the structure: the red feature, ideally the fruit of the plant, appears very late in the growing process, due to the high amount of loops between the lower part of this feature and the root of the animation. This characteristic, that generates a biologically correct evolution, isn't a singularity of the current example, but it can be expected in most plant models, where a long straight stem precedes more complex leaves, flowers and fruits.

A sigmoidal function (figure 9.1), actually contributes to have a more natural plant development and specifically it helps to remove the small "bumps", that occur when a feature reaches its maximum length. Although this effect can be seen only on live demos, figure 9.3 shows a set of screenshots for comparison purposes.



Video: http://www.youtube.com/watch?v=tE4oZ5rpRYQ

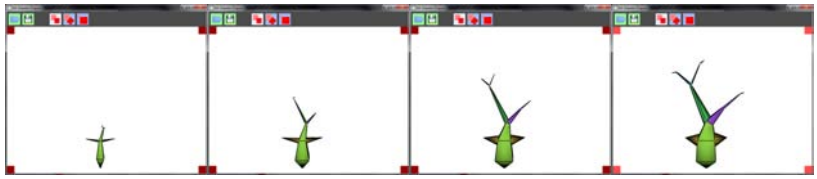
Figure 9.2: Morphing: growing plant with linear interpolation. Screenshot are not taken at constant intervals.

9.2 Gravity

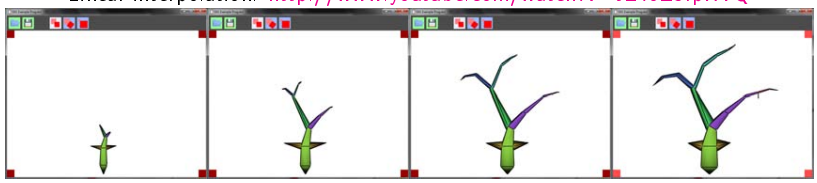
In order to test the capability of the system to support dynamic behaviors, a gravity modifier has been implemented. It basically applies the Newton's law of motion [Hol06] to each individual skeleton nodes in order to simulate the effect of the gravity on a semi rigid body. The simulation has been implemented in 3 steps:

Mathematical model: in the first step a mathematical model applies the Newton's Laws of motion to each individual skeleton node to simulate the behavior of the mesh under these forces.

Constrains of skeleton structure: as the mathematical model has been applied to each individual skeleton node individually, there is the possibility to affect also the length of each skeleton segment, which are intended to be rigid. A second step is therefore necessary to enforce a set of constrains and assure



Linear interpolation: http://www.youtube.com/watch?v=tE4oZ5rpRYQ



Sigmoidal interpolation: http://www.youtube.com/watch?v=v4KyYNLq0WM

Figure 9.3: Morphing: growing plant with sigmoidal (top) and linear (bottom) interpolation.

that the mesh maintains the intended shape.

Rendering: finally the position of the vertices are updated and the mesh is rendered.

9.2.1 Mathematical model

The mathematical model that affects each skeleton node includes:

- A constant force $Vec3f(0, g, 0)$ to simulate the effect of the gravity
- A proportional force $k\triangle l$ to simulate the reaction of the object to the gravity. This effect has been modeled with Hooke's law, where $\triangle l = (pos(n) - pos(n.parent)) - (pos_{nominal}(n) - pos_{nominal}(n.parent))$ is the difference between the current relative position and the nominal relative position of the same skeleton node.
- A inherited velocity component $\triangle v = pos(n) - pos_{old}(n)$ to simulate inertia (first Newton's law). A constant drag (0.1 to 0.8) has been applied to assure stability.

The final model therefore is:

$$pos(n) = drag * \triangle v + a * timeStep^2 \tag{9.1}$$

$$a = Vec3f(0, g, 0) + k\triangle l \tag{9.2}$$

9.2.2 Constrains of skeleton structure

One of the consequences to apply the mathematical model to each skeleton node individually, it is the possibility to affect also the length of each skeleton segment. Although the structure can blend under the gravity, the skeleton segments are intended to be rigid and they aren't supposed to change their length.

This can be enforced using the same relaxation technique that Thomas Jakobsen presented in its paper "Advanced Character Physics" [Jak01] to solve multiple constrains for a set of particles.

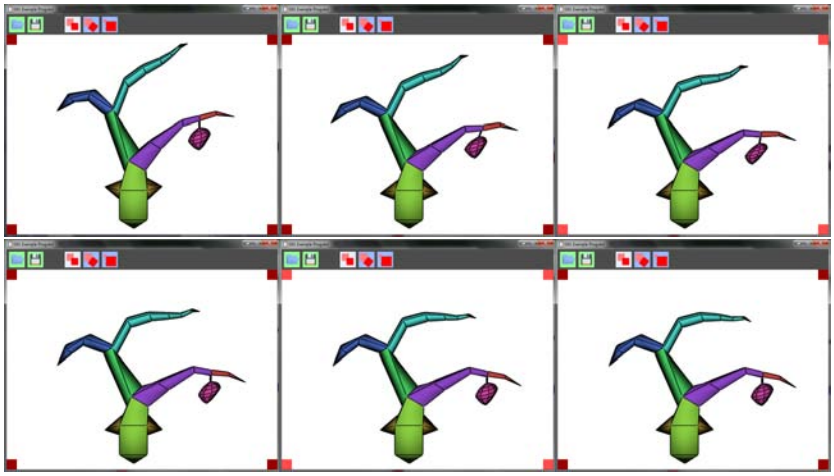
> "A common model for cloth consists of a simple system of interconnected springs and particles. However, it is not always trivial to solve the corresponding system of differential equations. It suffers from some of the same problems as the penalty-based systems: Strong springs leads to stiff systems of equations that lead to instability if only simple integration techniques are used, or at least bad performance – which leads to pain. Conversely, weak springs lead to elastically looking cloth.
>
> However, an interesting thing happens if we let the stiffness of the springs go to infinity: The system suddenly becomes solvable in a stable way with a very simple and fast approach."[Jak01]

The key idea here is iterating multiple times over each individual constrain until the system reaches the desired level of stability. One of the interesting aspect of this technique is that it always hits a good compromise between performances and quality: in the current implementation for example each constrain has been relaxed 10 time, a number that in most cases leads to good looking animations, but when it is insufficient the animation will look a bit less clean without any disruptive results both quality and performance wise.

9.2.3 Rendering

In the last phase, the new vertices positions are applied and the rendering is performed.

Although the physical simulation is actually performed before each rendering, the independence of this last step is a key prerequisite to decouple the actual dynamic model from its representation (rendering).



Video: http://www.youtube.com/watch?v=C_X6RI5rCs0

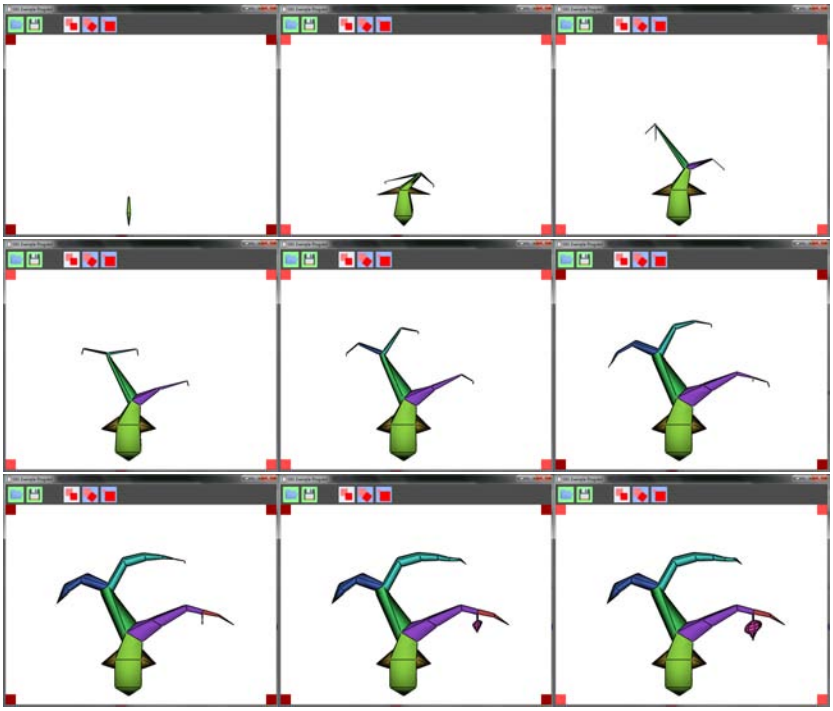
Figure 9.4: Gravity: plant. $Drag = 0.1$. Screenshot are taken at constant intervals.

9.3 Morphing and Gravity combined

The modularity of the animation system makes it possible to combine morphing and gravity behaviors, so that the gravity affects the plants while it is growing. The result (figure 9.5) isn't perfect, probably due to some side effects of the mathematical model used to emulate the gravity, but it is still impressive considering the generality of the algorithms used, those have no extensive knowledge regarding the skeleton structure.

9.4 Conclusions

An animation system tailored for polar meshes can surely exploit some of their features, but the concrete benefits seem to be more evident on bottom-up animations: if the desired movement can be defined with localized behaviors and they don't need



Video: http://www.youtube.com/watch?v=D6-2HqmIPUE

Figure 9.5: Morphing and Gravity combined: growing plant. $Drag = 0.5$. Screenshot are not taken at constant intervals.

an extensive knowledge over the entire skeleton structure to perform the deformation, then polar meshes provide the necessary support for a system that is both flexible and easy to use.

On the other hand there are scenarios where a global view on the entire skeleton is a fundamental prerequisite to design the animations and in these cases polar meshes don't have anything to add to the standard commercial tools: common techniques remains the best solution to deal with top-down movements, like character animations. Nonetheless an interesting solution can be reach combining traditional tools with a modeling system based on polar meshes: in this way the designer can procedurally generate the skeleton, exploiting the automatic skinning, before moving the object on a traditional animation tool.

Characters animations are the perfect example for this paradigm: the recent release of Mecanim [unia], the new Unity's animation technology, is partially changing the traditional character animation pipeline and it is pushing towards a decoupling

between the character design and the animations design, in such a way that it is now possible to apply a broad class of humanoid animations to any humanoid characters. In this context it is important to exploit the existing animation database and to do so it is necessary to export skinned characters with a skeleton as close as possible to the standardized one.

It is clearly possible to develop a tool for generating the traditional bones structure, for procedurally skinning the mesh to it and, with a bit more effort, to automatically simplify the skeleton structure to make it fits the standard avatar. This mixed approach, between polar meshes and traditional software, has already been suggested by some Unity game designers, and probably it is the best way to publish a first commercial tool strongly based on polar meshes.

Part III

Procedurally generated content

Chapter 10

L-Systems

10.1 Introduction to L-Systems

Lindenmayer systems or L-Systems are a mathematical description of the development process of organic structures. Originally developed by Lindenmayer in 1968 [Lin68], this method has been initially applied to model the behavior of simple multicellular organisms and later it became one of the key modeling tools to describe the growing process of plants and trees.

L-Systems represents the object as a string of symbols, which are iteratively replaced applying a set of production rules in order to make the object evolving [Lin04]. It is important to note that in each iteration all the symbol replacements are concurrently applied, to simulate the biological evolution of the organic model at cellular level. As illustrative example the production $A \rightarrow AB$ applied to the initial string $ABBAAB$ generates $ABBBABABB$ in output.

Technically L-Systems are strongly based on the formal language theory and they can be defined with the grammar $L =< V, w, P >$, characterized by:

- An alphabet V, where:
  - V^* is the set of words over V

  - V^+ is the set of nonempty words over V

- An initial axiom $w \in V^+$
- A finite set of production rules $P \in V \times V^*$, where each production $(a, X) \in P$ can be written as $a \to X$.

The success of this method isn't only due to its strong theoretical foundations, but it is also due to the several different geometric interpretations that have been developed during the years and that are currently used to get a graphical representation of the L-System. Nowadays a wise combination of complex geometrical features with L-Systems can produce "realistic visualizations of plant structures and developmental processes" [Lin04].

10.2 Classes of L-Systems

There are different classes of L-Systems, according to the expressiveness of their grammars:

- **D0L-Systems**, defined by a deterministic and context-free grammar, where each symbol appears at the left side in at most one production.
- **Stochastic L-systems**, based on a non-deterministic grammar, where for each symbol and for each iteration, it is possible to choose between multiple legal productions with the correspondent probabilities. This is used to generate different varieties of the same basic plant structure in different executions of the L-System.
- **Context-sensitive L-systems**, based on a context dependent grammar, where the interactions between different parts of the plant are taken into account, defining productions that depends, not only from the current symbol but also to its successors and predecessors.
- **Parametric L-systems**, based on parametric grammars, where the parameters are typically used to influence the graphical representation of the plant.

If the reader isn't familiar with these concepts, additional details regarding L-Systems can be found in [Lin04], while [Hop08] provides a good reference for grammars and formal languages.

10.3 Turtle representation of L-System

The traditional technique to visualize L-Systems is called turtle representation and it uses a reduced alphabet to describe the relative position between the element that represents the current symbol compared and the position of the previous symbol. A simple 2D example of this visualization technique is shown in figure 10.1, where the alphabet $V = \{F, +, -\}$ has been used to produce the string $FF + F - FF - F$. Each symbol of the alphabet is associated with a specific rendering operation:

- F means move forward for a fixed offset
- $+$ means rotate right
- $-$ means rotate left

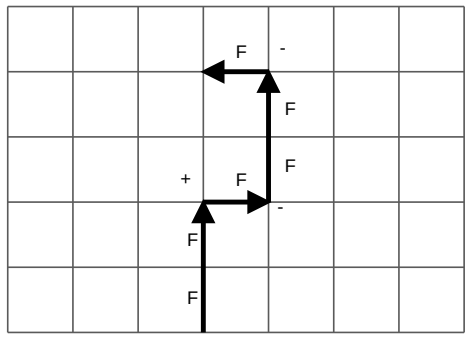


Figure 10.1: L-System: example of 2D Turtle Representation for the string $FF + F - FF - F$

The same technique can be applied in 3D spaces, simply defining an alphabet that includes:

- $+, -$ turning left and right
- $\&, \hat{}$ pitching up and down
- $\backslash, /$ rolling left and right

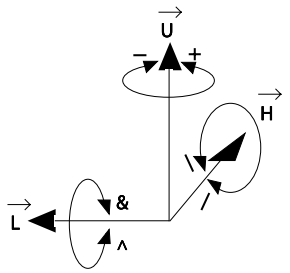
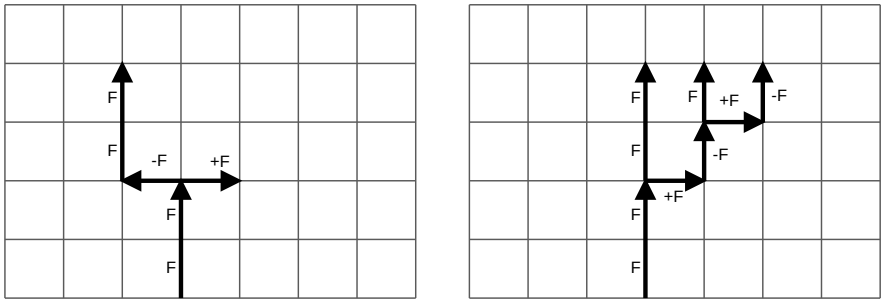


Figure 10.2: L-System: interpretation of alphabet for 3D turtle representation (source: [Lin04])

Finally, the turtle representation can be also expanded to support the branching structures of axial trees. This can be easily done using a stack and introducing two more symbols in the alphabet:

- [creates a branch at the current position, pushing the state on the stack.
-] moves the rendering back to the previous branch point, restoring the state from the stack (pop)

A couple of very simple 2D examples are shown in figure 10.3.



(a) String: $FF[+F][-F+FF]$ (b) String: $FF[+F-F[+F-F]F]FF$

Figure 10.3: L-System: examples of 2D Turtle Representation with branches

10.4 L-System and polar meshes

Polar meshes are a good graphical tool to visualize L-Systems and they can easily produce fairly complex graphical interpretations with a relative small effort: each segment of a turtle based L-System can be mapped into a different (set of) feature in the polar mesh, while the branching structure is ensured by the capability of procedurally generate nested features.

As it has been discussed in depth above, skeleton branches are strictly defined in polar meshes and they are implemented with a clear, deterministic and well organized mesh topology. This unique characteristic of polar meshes is a key factor to keep the rendering operation simple and ensuring an high quality mesh, independently of the relative angle between the skeleton segments involved in the branch.

Chapter 11

Implementation

In order to evaluate and test the usage of polar meshes in this frame, a context-free, stochastic and parametric L-System has been integrated in the prototype. The current implementation is intended to be a simple proof of concept and it is in facts affected by several limitations, the most critical one is maybe the lack of real time string parser for L-Systems that forces the production rules to be hard-coded in the prototype.

The L-System that has been implemented uses a turtle-like graphical representation, where each symbol describe the behavior of a specific skeleton segment:

- Alphabet $V = \{P, L, L_0, [^L, L_R,]\}$, where
  - P moves forward and draws a pole
  - L moves forward and draws a simple loop
  - L_0moves forward and draws a simple loop that can eventually be replaced (see productions)
  - $[^L$ moves forward, it pushes the state on the stack, and it creates a nested feature on the left, drawing a loop to support it.
  - $[_R$ moves forward, it pushes the state on the stack, and it creates a nested feature on the right, drawing a loop to support it.

  -] pops the state from the stack and it restores the previous position and direction

- Initial axiom: user defined $w \in V^+$
- Production rules:
  - $L(step) \xrightarrow{1} L(step)$
  - $P(step) \xrightarrow{1} P(step)$
  - $L_0(step) \overset{pr(step)-pr(step)^2}{\rightarrow} [^L L_0(step+1)L(step+1)P(step+1)]$
  - $L_0(step) \overset{pr(step)-pr(step)^2}{\rightarrow} [_R L_0(step+1)L(step+1)P(step+1)]$
  - $L_0(step) \overset{pr(step)^2}{\rightarrow} [^L L_0(step+1)L(step+1)P(step+1)][_R L_0(step+1)L(step+1)P(step+1)]$
  - $L_0(step) \overset{1-2*pr(step)+pr(step)^2}{\rightarrow} L(step)$

The probabilities are defined by the $pr(step)$ function, that is also used to prevent an unbounded growth of the plant. It is also important to notice how the probabilities sum to 1, defining a consistent grammar [odAtD94].

$$pr(step) = 1 - \frac{1}{1 + e^{5.2-step}} \tag{11.1}$$

Figure 11.1: L-System: probability function used to limit the growth process (Eq. 11.1)

Similar functions have also been applied to adjust length and thickness of the features from the root to the leafs of the plant (Eq. 11.2). Sigmoidal functions are good approximations of a wide range of biological growing processes, where "the initial part of the curve represents the growth of a young organism, while the latter part corresponds to the organism close to its final size" [Lin04] and they can be easily implemented exploiting the parametric nature of the underlying grammar.

$$\mathcal{L} = 1 - \frac{1}{1 + e^{2-step}} \tag{11.2}$$

11.1 Implicit string representation and real time mesh operations

Usually L-Systems produces an explicit string of symbols that is later converted in its graphical representation often with offline methods, but in this prototype it has been implemented a different approach: polar meshes contain by themselves enough structural information to completely avoid any explicit symbol representation and to apply the productions directly to the mesh, in real time. In this case the mesh itself can be considered the direct output of the L-System, where loops represent symbols and backbones represent the string structure.

In this framework, the productions have been coded as a sequence of polar mesh operations, a procedural application of the same modeling tools that have already been presented in the first part of this report. This is a powerful approach that can be used, in the future, to develop more advanced tools, where procedurally generated L-Systems may use variants of a manually defined feature.

Chapter 12

Analysis

Despite it is just a proof of concept, this prototype already produces fairly good results (Figure 12.1): the feature nesting is smooth and clean, it doesn't present any singularity and the level of details is adjusted in real-time, affecting only a localized area of the mesh and making sure that new features can always be added.

On the other hand, the prototype is affected by several limitations, due to both the current implementation as well as some specific characteristics of the method itself.

The first comment is regarding the current branching behavior that produces nested features always from the side of the parent feature, which doesn't continue to grow after the branch. This is a design choice that has been made because of its simplicity and it is the actual reason why the final plant looks more like a shrub than a tree, but it can be easily changed.

A deeper problem that isn't easy to fix instead, is the lack of direction: in order to nest a new feature on an existing loop, it is necessary to define which vertices of the loop will be affected by the nesting operation, but selecting the right vertices in order to have the gap oriented into a particular direction, isn't a trivial task and brute forcing seems to be the only way to chose an optimal solution.

It is interesting to notice how this problem isn't due to any specific detail of the current implementation but it is instead a consequence of the usage of polar meshes:

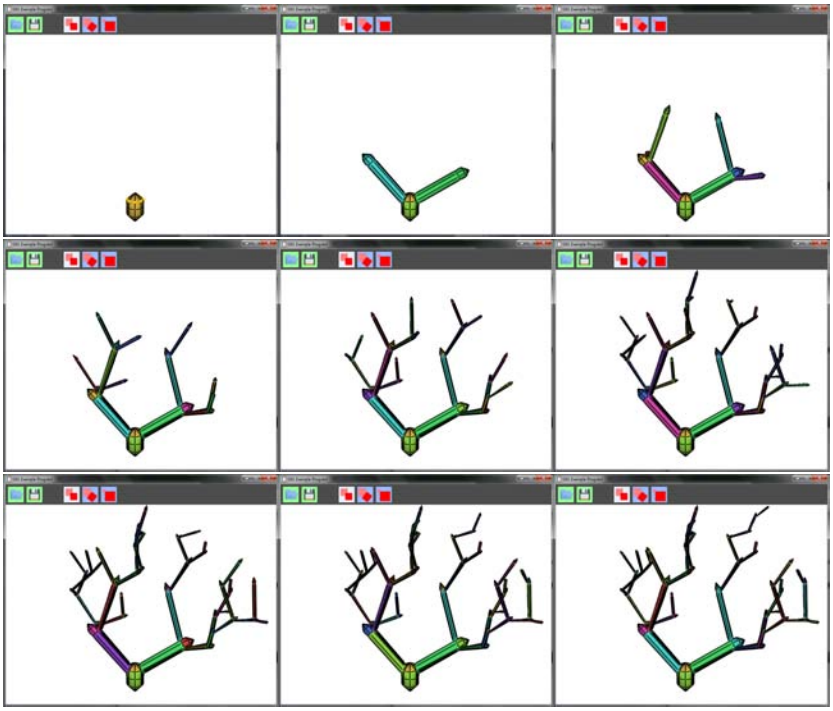


Figure 12.1: L-System: example.

they do support mesh operations that directly affect (or that are affected by) the skeleton, but any structural information needs to be inferred in real time and this is computationally heavy.

The system is, in facts, already affected by performance issues, that can be easily seen removing the stochastic branching limitation and allowing the generation of two nested features at each iteration. Figure 12.2 is an example that took several seconds to be computed and in 11 iterations it becomes so time consuming to reach the boundaries of what it is usually considered real-time.

The profiler used for a deeper analysis revealed that most of the effort is partitioned between the actual generation of the new features (20%) and the updates of the rendering data (44%), where most of the time (34% out of 44%) is actually spent computing the new per-vertex normals.

Although the code isn't completely optimized it is clear that a drastic reduction of the overall effort is impossible and the theory of computational complexity actually supports this conclusion: although each feature requires roughly the same effort

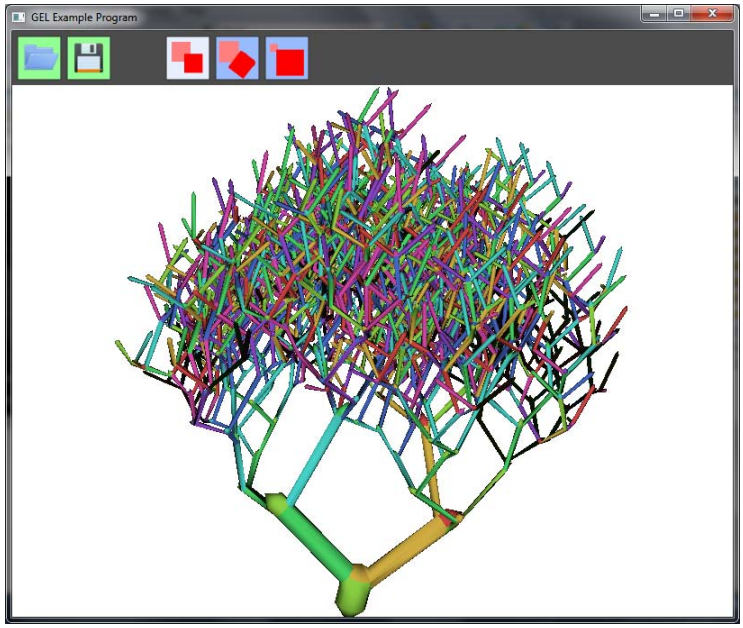


Figure 12.2: L-System: example of plant with performance issues (11 steps, 400K vertices)

(this is true regarding both creation time and rendering time), at each step, for each leaf, two new feature are created. This leads to a full binary tree with a depth proportional to the number of iterations applied and where all the nodes induces a constant effort. It is well known [CLRS01] that in this case a tree traversal will require an exponential complexity over the number of steps:

$$\sum_{i=0}^{steps} O(2^i) = O(2^{steps}) \tag{12.1}$$

All considered this example shows that it is definitely possible and convenient to use polar meshes to visualize stochastic and parametric L-Systems, although there are some challenges that a real production tool needs to solve, and this can be easily extended to context dependent L-Systems, providing the correct support and per-feature data.

Part IV

Conclusions

Chapter 13

Conclusions

As presented in the first part of the report, the main goal of this project was evaluating the potential of a modeling system based on polar meshes, in relation to modeling, animations and procedurally generated content.

Modeling

From the modeling point of view, this project can be seen as an evolution of SQM and it is interesting to compare the new system with this older method. In his thesis Mc Donnell [Don12] showed that SQM has a limited expressiveness and only a narrow range of shapes can be conveniently modeled with this technique. This is mainly because in SQM the final vertices positions aren't controlled by the designer but they are inferred from the skeleton structure.

In the current modeling system, these limitation have been completely overcome and it is now possible to model any arbitrary shape as long as it leads to a closed mesh. This has been reached allowing the user to define the exact position of each individual vertex (low level modeling) as well as with a deeper understanding of the concept of polar mesh: in the new system any combination of polar and annular regions is allowed and it leads to a mesh structure that is no longer based on a rooted tree (an important detail in the implementation of the SQM method) but

that now supports the generality of a non-simple graph (high level modeling).

This new modeling system can be definitely exploited to push the development of 3D assets into a more AGILE process and to some extent this is already supported in the prototype, but on the other hand the current implementation doesn't seem to provide a fair platform for a fast-paced designing process: the actual modeling still feels quite clumsy, mainly due to the poor GUI and limited low-level modeling tools. In this perspective an integration with a traditional 3D software can actually improve the user experience and provide a better support for the low level modeling.

Animations

One of the key characteristics of polar meshes is their implicit skeleton which can be inferred in real time from the topological structure of the mesh and that can be used to design more flexible animations. In this project a behavior based animation engine has been developed, where the mesh skeleton is automatically gathered and deformed according to the desired movement.

This approach turned out to be very effective for bottom-up animations, such as morphing, physically based motion and whenever the global movement is defined as a combination of local behaviors affecting each skeleton node individually; but it doesn't seem to help when an extensive knowledge on the entire skeleton structure is needed, like in humanoid animations. In these situations polar meshes still represent a valid modeling tool and they can always be used to procedurally generate the skeleton structure, but this topology doesn't lead to any real improvements animations wise.

Finally, polar meshes seems to provide a fairly good support for skinning algorithms, that typically correlate each vertex with skeleton nodes in a limited neighborhood, and further developments can actually provide a better overview of the potential of polar meshes in this sense.

Procedurally generated content

In principle a procedurally generated content is no more than a repeatable sequence of API calls, either related to modeling and/or animations. In this project it has been implemented an L-System to demonstrate the applicability of this principle to polar meshes and to evaluate to what extent polar mesh operations can be used as coherent and invokable API.

The outcomes show that it is definitely possible to automate sequences of mesh operations and polar meshes provide a really good support for procedurally generated content that massively affects the model structure: usually L-Systems are designed to produce an explicit string of symbols that is later converted in its graphical representation, but polar meshes contain enough structural information to be considered by themselves the direct representation of the L-system, which can apply the productions directly to the model. In this case the mesh itself can be considered the direct output of the rewriting rules, where loops represent symbols and backbones represent the string structure.

Finally, the L-System implemented in the prototype is based on context-free, stochastic and parametric grammars, which is a fairly broad and powerful class of formal languages, but that doesn't exploit the full potential of polar meshes. A further development in this direction should definitely consider to move towards a more general class of grammars, including context-dependent behaviors.

Chapter 14

Further Work

The project represents a positive proof of concept on the potential of polar meshes and it provides an overview across three lines of research: modeling, animation and procedurally generated content. The next step is clearly a deeper exploration in each of these areas specifically, a further development that carries on the academic investigation alongside to a more product oriented research.

In this project, thanks also to previous works, the concept of polar mesh reached a discrete maturity and the potential of a modeling system based on this topological pattern has been fundamentally proved. It is therefore time to evaluate its effectiveness with real world problems, designing how this modeling system can be delivered to the community and how it can be integrated with the large set of tools already available to CGI artists.

In this prospective it can be interesting to develop this tool as a plug-in for a popular 3D modeling software, such as Blender[1], 3DS Max[2] or Z-brush[3], in order to distribute it among the community, with a user interface that most designers are already familiar with and facilitating the integration of polar meshes into the assets production pipeline. An important part of this plug-in will also be the possibility to

[1] Blender: http://www.blender.org/
[2] Autodesk 3DS Max: http://www.autodesk.com/products/autodesk-3ds-max/
[3] Pixologic ZBrush http://pixologic.com/

automatically generate the bones structure associated with the implicit skeleton in order to export the skinned model to standard tools for further manipulation.

Part V

Appendices

Appendix A

Project management

A.1 Development process

This project has been developed between the 11 February and the 15 July 2013, using an incremental development process based on 22 iterations. In order to get the best out of the weekly Friday meetings, that provided feedback throughout the entire working process, each iteration started on Friday at noon and it finished one week later before the next meeting.

A short report, at the end of each iteration, summarized the weekly achievements and set the goals for the next 7 days. Part of these documents were also brief discussions on problems that were challenging the development of the thesis.

The preliminary plan, which has been made at the beginning of the thesis, focused mainly on the modeling aspect of the project and it turned out to be over pessimistic: at the beginning of May it was clear that it was running faster than expected and most of the risks associated with the development of the core set of operations weren't going to happen anymore. At the same time, a deeper understanding of the concept of polar mesh and a clearer vision of the entire thesis pushed the focus of the project towards animations and procedurally generated content and more intentional investigations have been planned in these directions.

These considerations caused an important redefinition of the original project plan and they are the reason of the current thesis structure, based on three distinct sections.

Week	From	To	Original Plan	Actual Plan
7	11-Feb	15-Feb	Study plan, Template, Problem definition	Study plan, Template, Problem definition
8	18-Feb	22-Feb	Basic prototype: mesh viewer	Basic prototype: mesh viewer
9	25-Feb	1-Mar	Operation list and data structures	Operation list and data structures
10	*4-Mar*	*8-Mar*	*Holiday*	*Holiday*
11	11-Mar	15-Mar	Loop based Operations	Risk analysis and thesis motivations
12	18-Mar	22-Mar	Operation: Add Feature	Loop operations and iterators
13	25-Mar	29-Mar	Operation: Add Feature	Core Operations
14	1-Apr	5-Apr	Expressiveness test	Report: Introduction
15	8-Apr	12-Apr	Operation: merging	VBO implementation, GUI, improved AddFeature
16	15-Apr	19-Apr	Operation: merging	Strictly and extended polar meshes
17	22-Apr	26-Apr	GUI Design	Expressiveness tests
18	29-Apr	3-May	Expressiveness test	**New Project plan**
19	6-May	10-May	GUI Implementation	Report reorganization
20	13-May	17-May	Data structure optimization	2 Level modeling
21	20-May	24-May	Tests and report	New introduction and topology considerations
22	27-May	31-May	Tests and report	Shader and VBO optimization
23	3-Jun	7-Jun	Results	Revised concept of polar mesh
24	10-Jun	14-Jun	Text polishing	Algorithm for skeleton recognition
25	17-Jun	21-Jun	Text polishing	Animations
26	24-Jun	28-Jun	Internal deadline	Procedural generated content
27	1-Jul	5-Jul		Documentation and Report
28	8-Jul	12-Jul		Text revision and appendices

Table A.1: Project plan

A.2 Risk analysis

At the beginning of this project an initial set of risks have been identified and evaluated with the standard "risk exposure" technique. A specific mitigation strategy have been consequently developed for the most critical ones (exposure > 1):

Risk	Prob	Impact	Effect	Mitigation strategy
The 3D modeling system seems useless	50%	5	2.50	Detailed description on short and long term benefits of the system
Under estimation of workload. Feasible work < 80%	80%	2	1.60	Close project management, per week goals
The 3D modeling system is ineffective	30%	4	1.20	Early feedbacks on the prototype
Under estimation of workload. Feasible work < 50%	20%	5	1.00	Close project management, per week goals
Unable to design 1 operation	40%	2	0.80	
Unable to implement 1 operation	60%	1	0.60	
Unable to design 2-3 operations	10%	5	0.50	
Unable to implement 2-3 operations	10%	3	0.30	
Unable to design more than 3 operations	5%	5	0.25	
Unsolvable technical problem on basic mesh editor	5%	5	0.25	
Unable to implement more than 3 operations	5%	4	0.20	

Table A.2: Project risks

Appendix B

Prototype

B.1 The shader

The shader is one of the most complex components of the current prototype and although it isn't part of core of this project, it is interesting from both a technical and a design prospective. The shader has been developed for two specific purposes:

Feedback: in order to understand and to feel comfortable with the modeling tool it is important that the user has real time feedback on the mesh structure, such as information regarding surface curvature, wireframe view, a clear distinction between loops and backbones, feedback on feature distribution along the model and of course a selection tool to highlight which part of the mesh will be affected in the current operation.

Performances: the traditional OpenGL approach to combine different information on the same image uses different rendering passes on top of each other. This approach intrinsically leads to a general degradation of the rendering performances and for mid size meshes it can already represent a problem for the visualization. On the other hand a customized shader can move to the GPU some of the complexity related to feedback visualization and it can keep a single rendering pass regardless the number of feedback involved.

The shader itself is therefore a combination of different features:

- Basic diffuse shader
- Quad based geometry shader and wireframe
- Color coded information

B.1.1 Basic diffuse shader

The light model that has been used is a basic per-fragment diffuse shader:

$$diffuse = fragColor * max(0, lightDirection \cdot normal) \tag{B.1}$$

$$gl\_FragColor = diffuse + ambient \tag{B.2}$$

B.1.2 Geometry shader and wireframe

It is well know that the geometry shader can be used to place a wireframe view on top of a solid shaded mesh but the method described in the OpenGL shading language cookbook [Wol11], based on the distance between the current fragment and the nearest edge, is typically applied to triangle based meshes and it becomes very tricky on quads.

The main reason of this increment of complexity is due to the triangulation performed in the geometry shader, that doesn't accept quads neither in input nor in output [geo]. The standard workaround uses "strip with adjacency" (so the mesh is rendered as a line strip) to produce a "triangle strip" of two elements for each quad in output, but in this way the fragment shader needs to draw the wireframe only near the edges that are on the external side of the original quad.

Although this method is still applicable, a more direct technique has been used in the prototype, where the wireframe has been rendered as a collection of long and thin additional quads, produced on the fly in the geometry shader. This is a variant of the silhouette method described in a famous blog post of Philip Rideout (http://prideout.net/blog/?p=54) and it provides a more direct control over the final result.

B.1.3 Color coded information

The final mesh color is completely procedurally generated and it is independent from the input values passed in the color buffer, which has been used to transfer a set of different information. An first interesting aspect is that although the color array contains per-vertex data, it has also been used to transfer per-halfedge and per-face information:

- The data regarding a specific halfedge is stored in correspondence of the vertex pointed by the halfedge itself.
- The data regarding a specific face is replicated and stored in correspondence of all the 4 vertices that define the face itself.

At this point packing a different number of feedback on the 3 color channels is a trivial task:

Red: in the R field it has been stored a numeric ID of the feature that the current face is part of.

Green: this field contains a boolean value to define whether the current edge has to be marked as a loop edge

Blue: in the blue channel a bit-mask has been used to specify if the current vertex, halfedge and/or face has been selected. As the value is a float the "bits" have been stored as peak values in correspondence of 0.1, 10 and 100.

The feature IDs have been used to procedurally generate the final mesh color, in a process designed to optimize readability and to emphasize the differences between connected features. To achieve this, the ids have been mapped to a limited set of different colors, equally spaced in hue spectrum, that is limited enough to make each color clearly distinguishable, but that is big enough to reduce the probability to use the same color for connected features.

Appendix C

Animation techniques

As briefly described in the dedicated chapter, animations can be defined in several different languages, each of them at different level of details. The most common methods [ana] to describe complex deformations are:

1. Keyframes
2. Animation scripting languages
3. Goal oriented motion
4. Motion capture

C.1 Keyframes

Keyframes is one of the oldest techniques in the history of computer graphics, originally developed in the Walt Disney studios [BW, Las87]. It is a fairly simple method where the designer defines the poses (vertices positions) in correspondence of specific time stamps, called keyframes, that the system linearly interpolates in order to approximate the pose in all the in-between frames.

Despite the obvious limitations of this technique, that requires a considerable amount of work in order to define complex animations, this method is still widely used for a large set of inorganic animations, especially in its improved form where the strictly linear interpolation has been replaced by more complex interpolation system, typically controlled by a Bèzier curve.

In figure C.1 for example, it is shown a simple animation where a cube has been translated and rotated in 3DS Max using the keyframes technique. Please note how the interpolation of the in-between frames can be controlled with the curves on the left side of the screenshot.

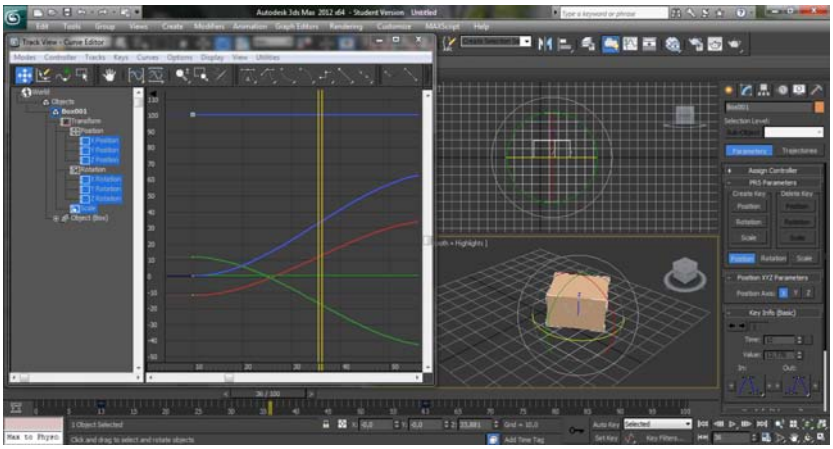


Figure C.1: Keyframe animation in Autodesk 3DS max 2012

C.2 Animation scripting languages

A complete different approach is offered by script engines, where the deformation is defined using descriptive languages. A typical example of animation scripting is the Improv System [PG96], developed by Ken Perlin and Athomas Goldberg at the Media Research Laboratory (New York University), where it is possible to define "real–time behavior–based animated actors" [PG96] using a English-like descriptive language. As shown in figure C.2, in the Improv system the descriptive code is first processed by the behavior engine to correctly instruct the internal animation engine to finally produce the right mesh deformation. An example of this method can be see in figure C.3, where the neutral face (left) has been deformed with a positive *smile* (center) and a negative *smile* (right).

Although the direct application of animation scripting languages has been fundamentally dismissed, replaced by more graphical inputs and techniques, the concept of a text/script representation of the animation is still widely used under the hood for several pug-in oriented animation systems.

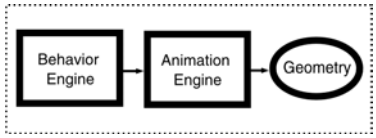


Figure C.2: Improv system: structure

Figure C.3: Improv system: example

C.3 Goal oriented motion

A method that works at an higher level of abstraction is Goal Oriented motion, where the animator specify the desired final behavior and the animation system compute the motion to reach or perform that behavior.

The most common example of goal oriented motion is Inverse Kinematic [ik-b], a technique that uses chains of connected joints. In this case the animator defines the position of the end-effectors at the end of the chains, while the system computes the position of each internal joints, congruently with the constrains.

Inverse kinematic is suitable to model rigid bodies and skeleton based characters, it is often combined with key-framing and with a dynamic description of the forces that act on the animated body.

Figure C.4: Unity3D: example of inverse kinematic [ik-a].

C.4 Motion capture

One of the most used animation techniques nowadays is motion capture, where a physical movement is tracked, recorded and coded into a virtual animation. In this case the deformation isn't designed in a virtual space but it is initially performed by an actor, a person or an object, who physically moves in the real world, while a specific system keeps track of the movements with dedicated hardware and software.

Motion capture is typically faster than traditional methods and it typically produces high quality animations: in many situations in facts the animation itself is complex, but it is naturally performed in the real world. A typical example is a walking cycle, that is extremely tricky to be defined in a virtual environment, but it is absolutely trivial to act.

The main disadvantage of motion capture is the cost of the tracking system, although several different solutions have been developed [ana]:

Optical systems: reflective markers are added on the actor and they are tracked by a set of fixed cameras (figure C.5). This system leaves the actor free to move as no cabling is required, but on the other hand the same actor can accidentally occlude some of the tracking point, creating gaps in the data-flow. This problem can be partially reduced with additional cameras, at the price of a considerable increment in the computational complexity.

Acoustic systems: a set of audio transmitters are attached to the actor's body, tracked by a set of audio receivers. This method doesn't suffer by occlusion problems but, on the other hand, the transmitters need to be wired and this can be a limitation for some animations. Additional limitations are also related to the maximum number of transmitters that it is possible to use and the accuracy of high speed moments tracking.

Magnetic systems: a third approach uses magnetic sensors, strapped on the actor, those are able to calculate "their spatial relationship to a centrally located transmitter" [ana]. Like with acoustic systems, this method requires wires that limits the freedom of movement of the actor, and it is affected by spacial limitations, but it doesn't suffer from occlusion problems.

Figure C.5: EUCROMA: example of optical motion capture system (http://eucroma.dk/)

Bibliography

[ana] anand@recoil.org. Animation techniques. http://3d.recoil.org/nojavascript/AnimTech.htm.

[ATC+08] Oscar Kin-Chung Au, Chiew-Lan Tai, Hung-Kuo Chu, Daniel Cohen-Or, and Tong-Yee Lee. Skeleton extraction by mesh contraction. *ACM Trans. Graph.*, 27(3), 2008.

[Bel] Adrian D. Bell. *Plant Form*. OXFORD UNIVERSITY PRESS. http://www.biodiversitylibrary.org/bibliography/53100.

[BW] N. Burtnyk and M. Wein. Computer-generated key-frame animation. *SMPTE*, 80(3):149–153.

[Cam] Marcus N Campbell. C++ cubic spline class. https://www.marcusbannerman.co.uk/index.php/home/latestarticles/42-articles/96-cubic-spline-class.html.

[Cas12] Thomas J. Cashman. Beyond catmull-clark: A survey of advances in subdivision surface methods. *COMPUTER GRAPHICS forum*, 31(1):42–61, 2012.

[CLRS01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, 2001.

[Dil] Kernon Dillon. 3d mesh topology tip: Quads vs. triangles. http://blendernewbies.blogspot.dk/2008/11/3d-mesh-topology-tip-quads-vs-triangles.html.

[DKM13] Sophie Viseur Dimitri Kudelski and Jean-Luc Mari. Skeleton extraction of vertex sets lying on arbitrary triangulated 3d meshes. *LNCS*, (7749):203–2014, 2013.

[Don12] Michael Mc Donnell. *Skeleton-based and Interactive 3D Modeling*. PhD thesis, Department of Informatics and Mathematical Modelling, Technical University of Denmark, 2012.

[Dra] Nick Dragan. Intro to physically-based deformable body modeling. www.cs.unc.edu/~lin/COMP768-F07/LEC/24a.ppt.

[geo] Opengl wiki: Geometry shader (core v4.3). http://www.opengl.org/wiki/Geometry_Shader.

[GPD] Callum Galbraith, Przemyslaw Prusinkiewicz, and Campbell Davidson. Goal oriented animation of plant development. In *10th Western Computer Graphics Symposium*, pages 19–32. Citeseer.

[HF01] Jim Highsmith and Martin Fowler. The agile manifesto. *Software Development Magazine*, 9(8):29–30, 2001.

[Hol06] S. Holzner. *Physics For Dummies*. Wiley, 2006.

[Hop08] J.E. Hopcroft. *Introduction To Automata Theory, Languages, And Computation, 3/E*. Pearson Education, 2008.

[Hor] Fred Horvat. Atari - jaguar explorer online. http://www.atariarchives.org/cfn/08/09/03/0616.php.

[htt] http://savingyoutime.wordpress.com. C++ matrix inversion (boost::ublas). http://savingyoutime.wordpress.com/2009/09/21/c-matrix-inversion-boostublas/f.

[ik-a] Unity - inverse kinematics. http://docs.unity3d.com/Documentation/Manual/InverseKinematics.html.

[ik-b] Wikipedia: Inverse kinematics. http://en.wikipedia.org/wiki/Inverse_kinematics.

[JAB12] K. Welnicka J. A. Bærentzen, M. K. Misztal. *Converting Skeletal Structures to Quad-Dominant Meshes*. PhD thesis, Department of Informatics and Mathematical Modelling, Technical University of Denmark, March 2012.

[Jak01] Thomas Jakobsen. Advanced character physics. In *Game Developers Converence Proceedings*, pages 383–401. CMP Media, Inc., 2001.

[JLW10] Zhongping Ji, Ligang Liu, and Yigang Wang. B-mesh: A modeling system for base meshes of 3d articulated shapes. *Computer Graphics Forum*, 29(7):2169–2177, 2010.

[JT05] Doug L. James and Christopher D. Twigg. Skinning mesh animations. *ACM Trans. Graph.*, 24(3):399–407, July 2005.

[jui] Juice it or lose it - a talk by martin jonasson and petri purho. http://www.youtube.com/watch?v=Fy0aCDmgnxg.

[Ket] Lutz Kettner. Cgal user and reference manual, halfedge data structures. release 4.1. http://www.cgal.org/Manual/latest/doc_html/cgal_manual/HalfedgeDS/Chapter_main.html.

[Kob00] Leif Kobbelt. $\sqrt{3}$-subdivision. *SIGGRAPH '00*, pages 103–112, 2000.

[KP07a] Kestutis Karciauskas and Jörg Peters. Surfaces with polar structure. *Computing*, 79(2-4):309–315, 2007.

[KP07b] Kęstutis Karčiauskas and Jörg Peters. Bicubic polar subdivision. *ACM Trans. Graph.*, 26(4), October 2007.

[Las87] John Lasseter. Principles of traditional animation applied to 3d computer animation. *SIGGRAPH Comput. Graph.*, 21(4):35–44, August 1987.

[LCF00] J. P. Lewis, Matt Cordner, and Nickson Fong. Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. *SIGGRAPH '00*, pages 165–172, 2000.

[Lin68] Aristid Lindenmayer. Mathematical models for cellular interaction in development: Parts i and ii. *Journal of Theoretical Biology*, 18, 1968.

[Lin04] Przemyslaw Prusinkiewicz Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. The Virtual Laboratory, 2004.

[LLP11] J. Houle L. Leblanc and P. Poulin. Modeling with blocks. *Vis. Comput.*, (27):555–563, 2011.

[Mot] Giovanni Motta. 3d geometrical transformations - foley and van dam, chapter 5. http://www.cs.brandeis.edu/~cs155/Lecture_07_6.pdf.

[odAtD94] Rieks op den Akker and Hugo ter Doest. Weakly restricted stochastic grammars. In *Proceedings of the 15th conference on Computational linguistics - Volume 2*, COLING '94, pages 929–934, Stroudsburg, PA, USA, 1994. Association for Computational Linguistics.

[PG96] Ken Perlin and Athomas Goldberg. Improv: a system for scripting interactive actors in virtual worlds. *SIGGRAPH '96*, pages 205–216, 1996.

[Ros] Rachel Rosmarin. Why gears of war costs $60. http://www.forbes.com/2006/12/19/ps3-xbox360-costs-tech-cx_rr_game06_1219expensivegames.html.

[Roy70] Walker W. Royce. Managing the development of large software systems: concepts and techniques. *Proc. IEEE WESTCON, Los Angeles*, pages 1–9, August 1970. Reprinted in *Proceedings* of the Ninth International Conference on Software Engineering, March 1987, pp. 328–338.

[ski] Ucsd, course 169: Computer animation - skinning. http://graphics.ucsd.edu/courses/cse169_w05/3-Skin.htm.

[So] Da So. Opengl beginning guide(glui). http://soda815.blogspot.dk/2012_08_01_archive.html.

[Sta] Eric Stallsworth. Why are video games so expensive? http://www.alteredgamer.com/free-pc-gaming/21118-why-are-video-games-so-expensive/.

[unia] Unity3d: Simple and powerful animation technology. http://unity3d.com/unity/animation/.

[unib] Unity3d: Time manager. http://docs.unity3d.com/Documentation/Components/class-TimeManager.html.

[Wik] Wikipedia. Subdivision surface. http://en.wikipedia.org/wiki/Subdivision_surfaceÂ\protect\T1\textsection.

[WLW07] Jinzhong Wu, Xuehui Liu, and Enhua Wu. Mesh deformation under skeleton-based detail-preservation. In *Computer-Aided Design and Computer Graphics, 2007 10th IEEE International Conference on*, pages 453–456, 2007.

[Wol11] D. Wolff. *OpenGL 4.0 Shading Language Cookbook: Over 60 Highly Focused, Practical Recipes to Maximize Your Use of the OpenGL Shading Language ; [quick Answers to Common Problems]*. Packt Pub Limited, 2011.