Teaching 3D Computer Animation to Non-programming Experts

Dan Casas[1]

[1]Universidad Rey Juan Carlos

Abstract

This paper describes a Computer Animation course aimed at novice Computer Science and Engineering students with minimal programming skills. We observe that students enrolled in Computer Graphics (and related) undergraduate degrees usually face a Computer Animation subject early in their programs, sometimes even before they develop strong software development and programming skills. This causes that assignments and tasks where students should focus on the Computer Animation concepts, end up in frustration and massive efforts to just get over-complicated developing frameworks running. Instead, we propose a Computer Animation course based on small MATLAB tasks that covers a large range of topics and it is adapted to students with minimal programming skills. For each topic, we provide a brief theoretical summary and links to fundamental literature, as well as a set of hands-on tasks with the necessary source code to get started. A user study shows that students who took this course were able to better focus on the fundamental concepts of the subject, circumventing the need to learn advanced programming skills. Course material is available on a public GitHub repository, and solutions are provided upon request from course tutors.

1. Introduction

Computer Animation is a fundamental topic in the area of Computer Graphics with a large number of applications, including visual effects, animated films, videogames, and robotics. Bachelor and Master students in Computer Science and related degrees usually face a Computer Animation subject early in their program, in some cases even before having (or while still learning) solid programming skills. Therefore, developing course content and assignments that do not require advanced programming skills or complicated frameworks is essential for efficient teaching. To this end, this paper proposes a Computer Animation course with programming assignments based on MATLAB [MAT20a], and circumvents the need for any prerequisite programming course. Our course enables fresh students to learn fundamental Computer Animation algorithms with hands-on exercises and almost no programming-related issues overhead. Course material, including assignments and demo codes, are available at the GitHub repository `http://github.com/dancasas/computer-animation-in-MATLAB` and assignment solutions will be available upon request from course tutors.

MATLAB is a numerical computing framework developed by MathWorks [MAT20a], and its programming language offers efficient matrix manipulation, off-the-shelf plotting of functions and data, straightforward implementation of algorithms, and easy creation of user interfaces. MATLAB uses its own interpreted programming language, and can use an interactive mathematical shell or execute text files that contain MATLAB code. Thanks to its straightforward use, it is nowadays very popular in education, specially for teaching linear algebra and numerical analysis, and also very popular among researchers in image processing [MAT20b].

In this paper, we show that MATLAB can also be used to teach 3D Computer Animation concepts. The key advantage over traditional frameworks (*e.g.,* OpenGL library used from C++) is twofold: first, MATLAB removes the need for manually managing the memory allocation required when using arrays and variables, because it is an scripting programming language. Memory managing is a common headache for novice Computer Science students, who usually get stuck allocating and freeing memory tasks instead of focusing with the goal of the task; and second, MATLAB removes the need to explicitly set up a 3D virtual environment to output the results, because it naturally provides a canvas to draw. Alternative frameworks (*e.g.,* based on OpenGL) require setting up a virtual camera and the associated matrices (*i.e.*, model, view, and projection matrices), which is also a common headache for novice students.

Notice that nowadays there are many 3D computer graphics applications (*e.g.*, Autodesk Maya, Blender) than *already* implement most of the concepts described in this course (*e.g.,* forward kinematics, human animation, curve interpolation, etc.). These tools could also be *used* by non-programming experts – however, the goal of this course is to *learn* the fundamental ideas of these concepts, not their applicability. Therefore, here by non-programming experts we refer to those who are in the process of learning how to program, not those who are basic users of animation tools (*e.g.,* artists).

DOI: 10.2312/eged.20211004

2. Background and Context

Engaging and meaningful assignments are fundamental to teach Computer Science in general, and even more important in Computer Graphics, as stressed by Cunningham [Cun00]. However, providing attractive 2D and 3D Computer Animation exercises, tasks, and assignments to novice students can be difficult due to the necessary programming requirements to tackle the tasks. In fact, many Computer Graphics course curricula, including the redesign proposed by Ackermann and Bach [AB15], project-based courses [MGJ06], and activity-led [AP09], include object-oriented programming (*e.g.*, C++) and 3D graphics pipeline (*e.g.,* modern GLSL shaders, OpenGL) parts that can hinder the engagement with the course of students without strong programming skills.

Several attempts exist to mitigate the need for low-level API knowledge in Computer Graphics. Schweitzer *et al.* [SBG10] identify this common issue and propose to use of Processing language in an introductory course at the United States Air Force Academy. Marchese [Mar98] propose to use standard spreadsheets to easily visualize concepts such as affine transformations. Similarly, Elyan [Ely12] proposes a practical and non mathematical approach that encourages students to become active learners. Fink *et al.* [FWW13] propose a Java-based framework to teach modern Computer Graphics (*e.g.*, GLSL shaders) using raster-level algorithms that are more practical since they employ higher-level APIs.

In this work we focus on 3D Computer Animation, an area present in all Computer Graphics curricula according to the study by Balreira *et al.* [BWF17]. Therefore, designing engaging tasks and assignments for this area of Computer Graphics is important. Peters and Anderson [PA14] argue that meaningful exercises should be Independent, Iterative, Incremental, and Integrative. In our course, we follow exactly the same principles and describe tasks for 3D Computer Animation: the proposed tasks are self-contained (*i.e.*, independent, they do not depend on third-party code) and a few tasks towards the end of the course are incremental (*i.e.,* they build on tasks done earlier in the course).

3. Course Description

This course described in this paper is offered to undergraduate students in Computer Science and related degrees (*e.g.,* Mathematics, Engineering, etc.) in the first year, although it can be adapted to postgraduate degrees as well. The course consists in 3 ECTS taught in 13 sessions of 2 hours. Each session was split into 1 hour of theory, where the theoretical concepts were introduced, and 1 hour of lab, where students use the proposed Tasks to practice the concepts.

The course is structure into three main blocks: (1) Interpolation (Sec. 4), which introduces fundamental concepts of interpolation, mostly focused on polynomial interpolation, and proposes assignments to implement and visualize different techniques, including Hermite interpolation, Lagrange Interpolation, Spline Interpolation, and Bézier curves; (2) Image Warping (Sec. 5), which focuses on image morphing in particular, and proposes assignments to learn to synthesize seamless transitions between images; and (3) Character Animation (Sec. 6), which focuses on animations achieved through kinematic chains, including forward and inverse kinematic techniques, as well as facial animation with blendshapes.

In total, we provide 17 tasks for the 3 blocks described above. For most of tasks we provide the necessary starting code to enable students with minimal programming skill successfully follow the course. We also provide pseudocode to facilitate the understanding of the algorithms to implement. Importantly, each task is a self-contained assignment that do not depend on external code, cumbersome setup, or any third party library. However, some tasks build on previous tasks developed by the students earlier in the course, following the Incremental and Integrative modality of assignments proposed by Peters and Anderson [PA14].

4. Interpolation

4.1. Cubic Hermite Interpolation

The first interpolation technique studied in this course is cubic Hermite interpolation. This technique consists on defining a set of control points (*i.e.*, starting and end points of the curve, and their derivatives) that are used to linearly interpolate a set of polynomial basis such that the resulting curve passes through the input pair of points. It is largely used in computer graphics and geometric modeling to define curves and trajectories.

Given a pair of argument values x_1, x_2, the corresponding function value $f(x_1)$ and $f(x_2)$, and the derivatives at each x_k, the cubic Hermite interpolator method finds a third-degree polynomial that fits into the input points with the input derivatives. More specifically, for input values $x_k = \{0,1\}$, assuming a cubic polynomial $f(t) = at^3 + bt^2 + ct + d$ with a derivative $f'(t) = 3at^2 + 2bt + c$, this yields to the following linear system in matrix form

$$\begin{bmatrix} f(0) \\ f(1) \\ f'(0) \\ f'(1) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}. \tag{1}$$

Solving Eq.1, we find the Hermite basis matrix $\mathbf{M}$

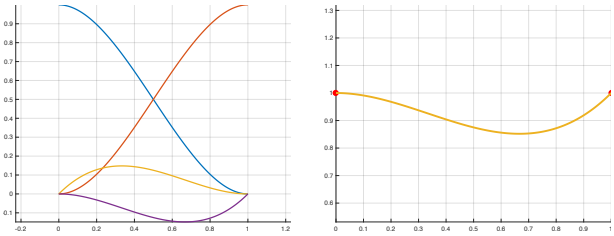
$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \underbrace{\begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}}_{\mathbf{M}} \begin{bmatrix} f(0) \\ f(1) \\ f'(0) \\ f'(1) \end{bmatrix} \tag{2}$$

from which, by plugin Equation 2 it into a cubic polynomial in matrix form, we can extract the Hermite basis functions

$$H_0(t) = 2t^3 - 3t^2 + 1 \tag{3}$$
$$H_1(t) = -3t^3 + 3t^2 \tag{4}$$
$$H_2(t) = t^3 - 2t^2 + t \tag{5}$$
$$H_3(t) = t^3 - t^2 \tag{6}$$

In order to let students practice with polynomial Basis and the linear combinations required to program Hermite interpolation, two tasks are proposed.

Task `1_1_plot_hermite_basis.m` gives the starting code to setup figures and plot polynomials in MATLAB with the function `plot()`, and shows how to plot H_0. Students are ask to plot the rest of Hermite basis, resulting in what is shown in Figure 1a. With this



(a) *Solution for Task 1.1: plot of Hermite basis functions.*

(b) *Task 1.2 with* $f(0) = 1$, $f'(0) = 0$, $f(1) = 1$, $f'(1) = 1$.

Figure 1: *Cubic Hermite interpolation.*

tasks, students familiarize themselves with basic MATLAB plotting functions.

Task `1_2_hermite_interpolation.m` asks to write a function that receives curve values and derivatives at time $t = \{0,1\}$, and compute the corresponding Hermite cubic polynomial. Students have to use the Hermite basis shown in Equation 2 to find the unknowns a,b,c,d, and then plot the resulting curve. Figure 1b shows the expected result with input values $f(0) = 1$, $f'(0) = 0$, $f(1) = 1$, $f'(1) = 1$. With this task, students familiarize themselves with basic operations to manipulate matrices and vectors.

4.2. Lagrange Interpolation

Lagrange interpolation is another method for polynomial interpolation. Given a set of points $\{x_{,k}, y_k\}$, this method finds the polynomial with the lowest degree that passes through the input points. Formally, the interpolation polynomial in a Lagrange form is a linear combination

$$L(x) = \sum_{j=0}^{k} y_j \ell_j(x), \tag{7}$$

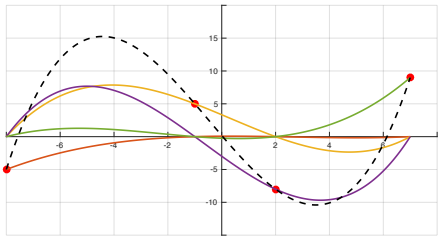
where $\ell_j(x)$ are the Lagrange basis polynomials defined as

$$\ell_j(x) = \prod_{\substack{0 \le m \le k \\ m \neq j}} \frac{x - x_m}{x_j - x_m}. \tag{8}$$

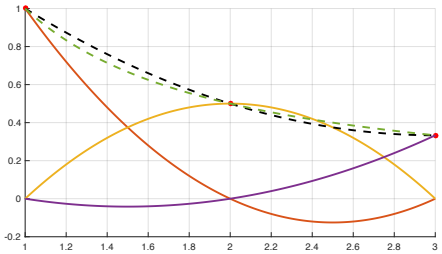
Notice that all basis polynomials are zero at $x = x_i$ except $\ell_i(x)$, for which it holds that $\ell_i(x_i) = 1$ because does not have the $(x - x_i)$ term.

Task `1_3_lagrange_interpolation.m` asks students to write a MATLAB function that receives as a input vectors `x` and `y`, which define a set of ponts $\{x_{,k}, y_k\}$, and compute and plot the corresponding Lagrange polynomial basis and the resulting Lagrange polynomial interpolation. Figure 2a shows the expected output for input point set $x = \{-8,-1,2,7\}$ and $y = \{-5,5,-8,9\}$. The Lagrange interpolator $L(x)$ is depicted in dashed black line, each of the polynomial basis ℓ_j in solid color lines, and the input set in red solid circles. Notice that all $\ell_j(x)$ are 0 for all x_k except one (*e.i.*, when a basis passes through a red circle, the rest of basis are 0), which is the main property of the Lagrange polynomial basis.

Task `1_4_lagrange_from_sampled_function.m` asks students to pick a function $f(x)$, plot it, and sample 3 points to obtain a set of $\{x_{,k}, y_k\}$. Then, similar to Task 1.3, they need to find



(a) *Task 1.3. In dashed black, the Lagrange interpolator for point set* $x = \{-8,-1,2,7\}$ *and* $y = \{-5,5,-8,9\}$.



(b) *Task 1.4 with* $f(x) = 1/x$, $x = \{1,2,3\}$

Figure 2: *Lagrange interpolation*

a Lagrange interpolation polynomial $L(x)$ that passes through the sampled points, and plot it. As a result of this tasks, and as shown in Figure 2b, students should notice that $f(x)$ (in dashed light green) and $L(x)$ (in dashed black) are not exactly the same curve, but they do overlap perfectly at the sample points (in solid red circle).

4.3. Quadratic Spline Interpolation

Spine interpolation is a form of interpolation based on piecewise polynomials. It is a method often preferred over Lagrange interpolation when there is a large number of sample points, because it keep polynomial degree low, which prevents undesired oscillations. The method consists in fitting a quadratic polynomial within each interval of the sample points, and enforce continuous first and second derivatives at the knots (*e.i.*, sample points).

To find the quadratic curves, having a set of input points $\{x_k, y_k\}_{k=1}^{K}$, we will define a piecewise spline $S(x)$ consisting on $K-1$ quadratic polynomials, one per each segment. Assuming the quadratic polynomial form $f_i(x) = a_i x^2 + b_i x + c_i$ and its derivative $f_i'(x) = 2a_i x + b_i$, we can set up a linear system of equations with $3(K-1)$ unknowns and $3(K-1)-1$ equations. To solve such undetermined system, we can enforce one more constraint, for example $f_0'(x_0) = 0$, which enforces the derivative of the starting point of the curve to be 0, or $a_0 = 0$, which enforces the first curve to be straight line.

Task `1_5_quadratic_spline.m` asks to write a MATLAB function that receives a set of data points $\{x_k, y_k\}_{k=1}^{K}$, and finds the $K-1$ quadratic polynomials that form a spline $S(x)$ that passes through the input points. Students are asked to automatically formulate the required linear system in a matrix form, and solve it using the `linsolve` function or the operator `\`,

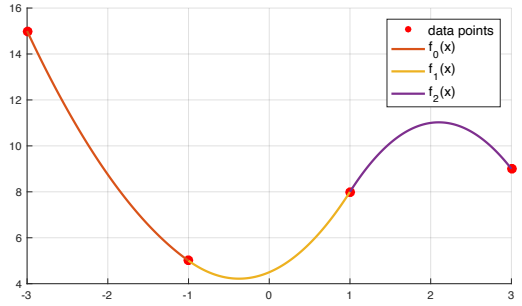


Figure 3: *Task 1.5, quadratic spline interpolation.*

which are used to solve a linear system of linear equations $\mathbf{Ax} = \mathbf{B}$. Figure 3 shows the expected results for input point set $\{(-3,15),(-1,5),(1,8),(3,9)\}$, depicting the three quadratic polynomials $f_0(x)$, $f_1(x)$, $f_2(x)$ in solid color lines. Notice how the link points between segments have a smooth curvature, thanks to the constraint on the derivatives on this points set in the system.

4.4. Bézier Curves

A Bézier curve is a parametric curve that leverages the Bernstein polynomials as a basis, and it is defined by

$$\mathbf{B}(t) = \sum_{i=0}^{n} b_{i,n}(t)\mathbf{P}_i, \quad 0 \leq t \leq 1 \tag{9}$$

where n is the number of basis (and the degree of the curve), $\mathbf{P}_i$ the i^{th} control point, and $b_{i,n}(t)$ the function basis. In contrast to the previous interpolation methods seen in this paper, a Bézier curve does not pass through all the control points (or data points). The first and last control points are always the end points of the curve, and the intermediate control points (if any) generally do not lie on the curve but control the shape of the curve.

Most common Bézier curves used in Computer Animation are quadratic and cubic (*e.i.*, degree 2 and 3 in Equation 9), which result in the expressions

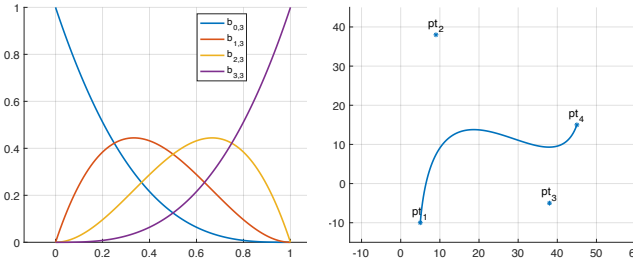
$$\mathbf{B}(t) = (1-t)^2\mathbf{P}_0 + 2(1-t)t\mathbf{P}_1 + t^2\mathbf{P}_2\,, \tag{10}$$

$$\mathbf{B}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2t\mathbf{P}_1 + 3(1-t)t^2\mathbf{P}_2 + t^3\mathbf{P}_3\,, \tag{11}$$

respectively. In order to practice with Bézier curve, four tasks are proposed below.

Task `1_6_compute_bezier_basis.m` asks students to write a MATLAB function that receives a parameter n and computes the Bézier basis required for a curve with this degree (*e.i.*, $n+1$ control points). To this end, students must write a generic function that implements the Bernstein basis polynomials. Figure 4a shows the plotted results for $n = 3$.

Task `1_7_plot_cubic_bezier.m` asks students to write a MATLAB function to compute and plot a cubic Bézier curve, following the expression in Equation 11. In order to get them started, a basic code on how to plot a linear and quadratic Bézier is given. Figure 4b shows the results of computing a curve with control points $\mathbf{P}_0 = (5,-10)$, $\mathbf{P}_1 = (9,38)$, $\mathbf{P}_2 = (38,-5)$, and $\mathbf{P}_3 = (45,15)$.



(a) *Solution for Task 1.6: plot of Bézier basis functions.* **(b)** *Task 1.7, a cubic Bézier curve.*

Figure 4: *Bézier.*

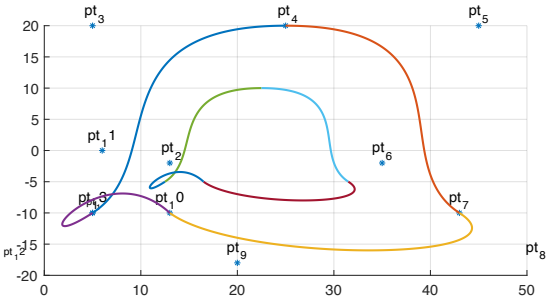


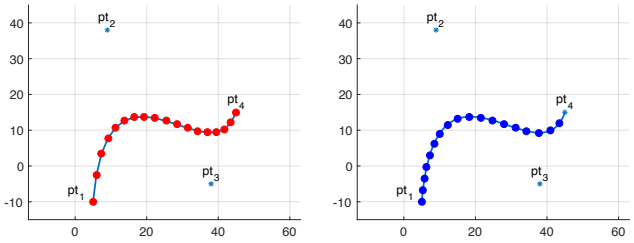
Figure 5: *Task 1.8, joint Bézier segments.*

Task `1_8_plot_bezier_segments.m` asks to write a function that computes and plots smooth joint segments of Bézier curves. In order to ensure that the curves are smoothly connected, students need to figure out how to enforce that the derivative of the end point of a curve is the same as the derivative of the starting point of the next curve. Figure 5 shows an example of this, where 4 curves are smoothly joined to create a closed shape. Additionally, this task also asks to apply rigid transformations to the control points $\mathbf{P}_i$ and plot the resulting curve. This is also shown in Figure 5, where the inner curve is the result of translating and scaling the control points of the outer curve.

Task `1_9_plot_bezier_equidistant_points.m` is about finding equidistant points along a Bézier curve. This cannot be naively done using a fix time interval Δt, which results in non-equidistant points over the curve as shown in Figure 6a. Instead, students must fist compute the length of the curve, something that cannot be done analytically. Therefore, an auxiliary function `compute_arc_length()` must be written to approximate the length of a Bézier curve using a forward differences approach (*e.i.*, compute $\sum ||\mathbf{B}(t) - \mathbf{B}(t+\Delta t)||$). Once the length is known, equidistance points can be plotted along the curve, as shown in Figure 6b.

5. Image Warping

Image warping [GM98] is the process of manipulating digital images to distort the original content. Straightforward warping operations include translation, rotation, and scale of images but, specially



(a) *Task 1.9. Using constant a Δt to evaluate a Bézier $\mathbf{B}(t)$ does not result in equidistant points.*

(b) *Task 1.9, equidistant points over a cubic Bézier curve.*

Figure 6: *Plot equidistant points over Bézier arc.*

in the area of Computer Animation, other effects such as image morphing are studied.

Morphing is a special effect or animation that distorts one image or shape into another through a seamless transition. To compute a morphing effect between to images, first, we need to specify a set of correspondences between the images. Such set is then used to triangulate the image, *i.e.,* to build a 2D mesh using a Delaunay triangulation approach. See Figure 9 left and right columns for examples of 2D mesh in source (green triangle) and target (orange triangle) images, respectively. Then, we need to find the affine transformations $\mathbf{A}_i$ that convert each triangle $\mathbf{t}_i^{\text{src}}$ of the 2D mesh in the source image into the corresponding triangle $\mathbf{t}_i^{\text{dst}}$ in the target image. $\mathbf{A}_i$ can be easily find since we know the triangle vertices both in source and target, $\mathbf{T}_i^{\text{src}}\mathbf{A}_i = \mathbf{T}_i^{\text{dst}}$, where $\mathbf{T}_i$ is a 3×3 matrix of vertices of the i^{th} triangle. Once all the $\mathbf{A}_i$ are computed, we are ready to perform a per-pixel operation to warp the image. To this end, two alternative techniques exists: forward warping and backward warping.

Forward warping computes for each pixel (u,v) in the source image its corresponding pixel (x,y) in the destination image. This is done by applying to the source pixel the affine transformation A_i of the triangle where the pixel belongs. As summarized in the pseudocode show in Figure 7, this populates the destination image with pixels from the source image. The main limitation of this strategy occurs when the destination triangle is larger than the source, and therefore not all pixels (x,y) have a corresponding (u,v).

```
forward_warping(src, dst, A)
{
  for (u=0; y<src.height; y++) {
    for (v=0; x<src.width; x++) {
      (x,y)=A(u,v);
      dst[x,y]=src[u,v];
    }
  }
}
```

Figure 7: *Pseudocode for forward warping.*

Backward warping circumvents the limitation of forward warping by computing for each pixel of the *destination* image its corresponding pixel in the source image. This inverse operation, summarized in pseudocode in Figure 8, guarantees that all pixels of destination image are populated. In general backward warping is always preferred over forward warping.

```
backward_warping(src, dst, A)
{
  for (x=0; x<dst.height; y'++) {
    for (y=0; y<dst.width; x'++) {
      (u,v)=inv(A)(x,y);
      dst[x,y]=src[u,v];
    }
  }
}
```

Figure 8: *Pseudocode for backward warping.*

To create smooth and seamless transitions between to images, we need to warp both images to in-between locations, and then linearly interpolate the pixel values of the warped images. Figure 9 shows a toy example where we create a morphing animation between a green triangle (source) and orange triangle (destination).

In order to practice image warping, and image morphing in particular, 3 tasks are proposed.

Task `2_1_warp_forward.m` asks students to implement the forward warping technique discussed above and in Figure 7. To quickly get started, we provide all the necessary code to load and write images in MATLAB, an interactive interface to click and select keypoints with `ginput()` function, as well as an example of the usage of the `delaunay()` function to triangulate points. Toy images from 9 are provided, and students need to warp the the source image into the destination, and vice versa. Students need to report what are the artifacts that they observe due to employment a forward warping technique (hint: not all pixels in the destination image will be populated).

Task `2_2_warp_backward.m` asks to implement the backward warping approach summarized in 8. Analogous to Task `2_1`, they are given the functionality to read and write images, and select and triangulate keypoints. Students need to comment on the quality of the results, and why warped images overcome the artifacts from the previous task (hint: all pixels in the warped image are populated).

Task `2_3_incremental_warp_backward.m` asks to create seamless transitions between two images. To this end, students need to combine a backward warping approach to independently warp the source and target image to an in-between point, and then linearly interpolate the warped pixel values according. As a result, students will be able to create seamless animations beteeen two images, as shown in Figure 10. To ease the programming burden, we provide with the MATLAB functionality to create animated GIFs images with the resulting frames.

6. Character Animation

6.1. Kinematics

Kinematics is the study of the motion of points, objects, and groups of objects without considering the causes of its motion. Kinematics are used in Computer Animation to pose articulated characters

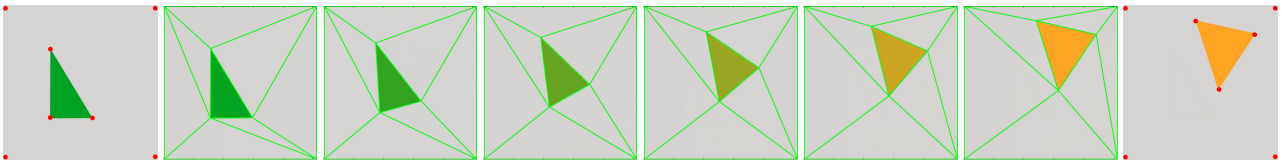


Figure 9: *Image morphing for a toy example, as proposed in Task 2.3. Left and right figures show the source and destination images and, in solid red circles, the selected keypoints. In-between figures show the 2D mesh computed from the keypoints, and how it is used to drive the per-pixel image warping, producing a smooth transition from source to target image.*

Figure 10: *Morphing images of faces in Task 2.3. Left and and right columns are source and target images, extracted from the CelebAMask-HQ dataset [LLWL20]. In between columns are warped (computed using the backward warping technique summarized in Figure 8) and blended images, producing a smooth interpolation from source to target.*

described by a set of rigid bodies connected by joints. See [SM09] Chapter 17.4 for a overview of this topic.

6.1.1. Forward Kinematics

Forward kinematics aims at finding the position of the end efector of a kinematic chain, given the configuration (*e.i.*, parameter values) of their joints. In other words, it addressed the problem $\mathbf{p} = f(\theta)$, where θ are the joint angles, and $\mathbf{p}$ the position of the end effector.

For an articulated character with a serial chain of N links, with join parameters θ_i, the kinematics equation, expressed in Denavit-Hartenberg notation, is defined as

$$ {}^{0}\mathbf{T}_N = \prod_{i=1}^{N} {}^{i-1}\mathbf{T}_i(\theta_i), \tag{12} $$

where ${}^{i-1}\mathbf{T}_i(\theta_i)$ is the affine transformation matrix from link $i-1$ to i. In 2D, such transformation matrices $\mathbf{T}_i$ are build up from rotation $\mathbf{R}_i$ and translation $\mathbf{Z}_i$ affine transformation matrices ${}^{i-1}\mathbf{T}_i(\theta_i) = \mathbf{R}_i\mathbf{Z}_i$. See [Kay05] for an in-depth introduction to forward kinematics, and further details about this expressions.

Task `3_1_plot_skeleton.m` asks to implement a MATLAB function to compute and plot the position of a 2D kinematic chain, defined by N rigid body parts of length l_n, linked with 1 degree of freedom (DOF) rotational joint parameterized by an angle θ_n. As a starting point, we are providing with the necessary code to plot a 1 DOF chain, including a structure to store a kinematic chain, basic matrix multiplication code to apply affine transformations $\mathbf{T}_i$

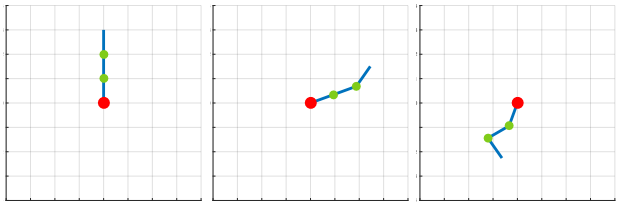


Figure 11: *Forward kinematics expected results from Task 3.1. Rest pose (left),* $\theta = [-70, 0, 35]$ *(center), and* $\theta = [160, -40, 95]$ *(right).*

following Equation 12, and plotting commands to visualized the resulting articulated chain. Figure 11 shows example results for this task, for a kinematic chain with 3DOFs, rigidly attached to the center of the grid.

6.1.2. Inverse Kinematics

Inverse kinematics aims at finding the angle parameterization for a kinematic chain such that the end effector position $\mathbf{p}$ reaches a desired goal position $\mathbf{g}$. In other words, we seek to find the inverse of the function f used in forward kinematics, $\theta = f^{-1}(\mathbf{p})$. In the area of Computer Animation, inverse kinematics is a important problem heavily used in videogames and visual effects, for example, to naturally pose human characters to reach a certain position with their hand or feet.

While forward kinematics is a well defined problem with an analytic solution, inverse kinematics is a lot harder, and it is generally tackled with an iterative method to find an approximated solution. In this course, we propose to solve it with the Jacobian inverse method, which is shown in pseudocode in Figure 12. In the rest of this section we provide a summarized description of the method, but students should check the extensive survey by Aristidou *et al.* [ALCS18] for detailed explanation of this and alternative methods.

Given a kinematic chain parameterized by joint angles θ, the Jacobian $\mathbf{J}$ is the matrix of partial derivatives of chain end effector $\mathbf{p}$ with respect to θ. The Jacobian iterative method find the θ values that bring $\mathbf{p}$ close to the target point $\mathbf{g}$ by repeatedly updating the values of θ with some increment $\Delta\theta$. Such increment is found by computing

$$ \Delta\theta = \alpha\, \mathbf{J}^{+}\mathbf{e}, \tag{13} $$

where $\mathbf{e} = \mathbf{p} - \mathbf{g}$ is the vector that points towards the goal from the current chain end effector, and $\mathbf{J}^{+}$ the pseudoinverse of the Jacobian. Note that all entries of $\mathbf{J}$ can be approximated numerically by

modifying slighly the current values of each θ_i and observing how the position $\mathbf{p}$ changes. See Section 5 of Aristidou *et al.* [ALCS18] survey or the techical report [Bus04] for details.

Task `3_2_inverse_kinematics_2D.m` asks to implement an inverse kinematic solution based on the Jacobian method. The iterative loop is summarized in the pseudocode from Figure 12. To define the skeleton structure and compute the position of the end effector $\mathbf{p}$ students can leverage all the code that was used for Task 3.1. Additionally, we also provide the necessary code to automatically update the plot in a MATLAB figure, given a new configuration of the angles θ, as well as the code to randomized the position of the goal $\mathbf{g}$ within a specific range. Figure 13 shows an example of the expected result from this task, where the kinematic chain anchored at location (0,0) smoothly reaches the goal position (shown in cross orange).

Task `3_3_inverse_kinematics_3D.m` is about extending Task 3.2 to a three dimensional scenario. The overall algorithm should follow the same structure depicted in the pseudocode from Figure 12. Figure 14 shows an example animation produced with this task, where a 3D kinematic chain with 3 rotational joints with 3 DOFs each, anchored at location (0,0,0), reaches the goal location shown in orange.

```
while(dist(skel.pos(), target) > threshold)
{
  J = compte_jacobian(skel)
  delta_rots = inv(J) * (target - skel.pos())
  skel.rots += delta_rots
}
```

Figure 12: *Pseudocode for Inverse Kinematics.*

6.2. Facial Animation

Facial animation is generally tackled in Computer Animation with a *blendshape* model [LAR*14]. Blendshape models are able to generate facial expressions as a linear combination of a set of basic expressions represented with 3D meshes. By changing the weights of the linear combination, a large variety of facial expressions and animations can be generate with little effort. This approach to model facial expressions has two main advantages: it implicitly provides a semantic parameterization (*e.i.*, the weights have intuitive meanings to animators); and it forces animators to stay within a plausible deformation stay (*e.i.*, no unrealistic expressions are produced).

More formally, a facial expression $\mathbf{f}$ can be computed as

$$\mathbf{f} = \sum_{k=0}^{K} \mathbf{b}_k w_k, \tag{14}$$

where $\mathbf{b}_k$ is a basic expression (*e.g.*, open mouth, smile, raise left eyebrow, etc.), and w_k the associated weight. However, Equation 14 is severely limited because each expression affects whole face. In other words, it is a global operation, and cannot operate just on local areas. This causes issues when, for example, $\sum w_k > 1$ which produces an undesired scaling factor. This is mitigated with the *delta blendshape formulation*, the approach largely used in the industry and in most of professional animation tools.

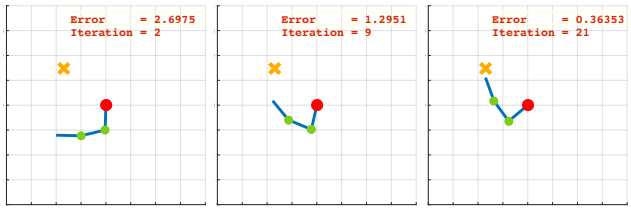


Figure 13: *Expected results from Task 3.2. A kinematic chain anchored at the center of the grid (red circle is the root) smoothly reaches the goal (in orange) using inverse kinematics.*

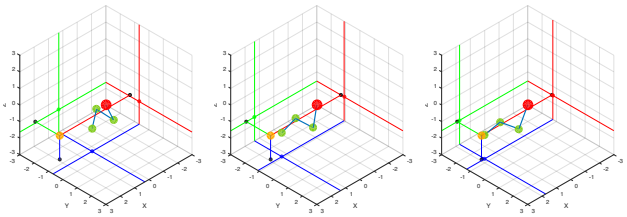


Figure 14: *Expected results from Task 3.3. A 3D kinematic chain anchored at the center of the 3D grid (red circle is the root) smoothly reaches the goal (in orange) using inverse kinematics.*

Delta blendshape defines a face model $\mathbf{b}_0$ as a neutral expression, and the rest of expressions $\mathbf{b}_k$ are updated with the difference w.r.t the neutral expression $\mathbf{b}_k - \mathbf{b}_0$. Therefore, Equation 14 is updated as

$$\mathbf{f} = \mathbf{b}_0 + \sum_{k=0}^{K} w_k(\mathbf{b}_k - \mathbf{b}_0). \tag{15}$$

Task `3_4_compute_global_blendshape.m` provides the basic functionality to load meshes stored in obj files, as well as a set of example expressions (also attached as a supplementary material). To goal of this task is to implement the global blendshape method defined in Equation 14. Students need to comment and report on what happens when $\sum w_k > 1$.

Task `3_5_compute_delta_blendshape.m` propose to start from the same initial code as in Task 3.4, but implement the delta blendshape approach defined in Equation 15 instead. Students should demonstrate that with this approach exaggerated expressions are possible without suffering from undesired global scaling effects.

7. Evaluation

To evaluate the proposed methodology, we asked 18 students to rate the course, on scale 1 to 5, according to different aspects. For each question, we compare the average score for this course with the average score for other animation courses of the same degree that use different programming languages (*e.g.,* C++). Figure 15 presents the results of this user study.

We first asked how much *the proposed methodology helps to successfully understand and focus on the fundamental concepts of the course* (*i.e.*, using MATLAB instead of less intuitive programming

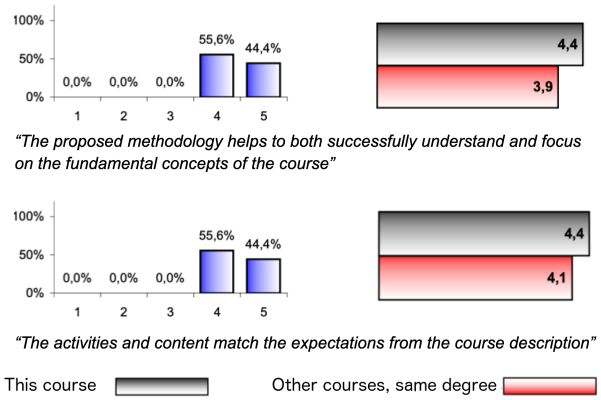


Figure 15: *User study of our methodology. Participants are asked their preference for the described methodology (top), and how much the content fits their expectation (bottom). For each question we provide histogram of scores 1 to 5 (left, in blue), and compare the average score of our methodology (in black) to the average score of other courses (in red) that use other programming languages (right).*

language), which resulted in an average score of 4.4. Notice that the same question about the methodology, asked to the same group of students in other courses of the same degree, resulted in an average score of 3.9. This hints that using programming languages that require deeper coding skills hinder the learning of the course fundamentals. In contrast, in the paper we propose on clear and self-contained programming tasks that allow students focus on what is being taught other than programming low-level details.

We then asked students to rate the content of the course, and how much did it fit to their initial expectations, given the information that was available when they sign up for the degree. Specifically, we ask how much *the activities and content match what they expect from the course description*, which resulted in an average score of 4.4. When the same question was asked for other courses of the same degree, the average score was 4.1. This indicates that the proposed methodology does not cause any distraction or divergence from the actual course content.

8. Conclusions

We have described a methodology to teach Computer Animation to non-programming experts. The need for such course arises from the fact that undergraduate students in Computer Science (CS) degrees typically face a Computer Animation (or related) course before developing strong programming skills. Thanks to the MATLAB-based self-contained assignments and tasks described in this paper, the students are capable of learning the basic principles of the field without getting stuck into programming-specific problems. Our user study with a group of students that passed this course demonstrate that they were able to better learn the key concepts of the subject, comparing to other courses where more complex programming languages are used.

Acknowledgments. The work was funded in part by the Spanish Ministry of Science (project RTI2018-098694-B-I00 VizLearning).

References

[AB15] ACKERMANN P., BACH T.: Redesign of an Introductory Computer Graphics Course. In *Eurographics - Education Papers* (2015). doi:10.2312/eged.20151021. 2

[ALCS18] ARISTIDOU A., LASENBY J., CHRYSANTHOU Y., SHAMIR A.: Inverse Kinematics Techniques in Computer Graphics: A Survey. *Computer Graphics Forum 37*, 6 (2018), 35–58. doi:10.1111/cgf.13310. 6, 7

[AP09] ANDERSON E. F., PETERS C. E.: On the Provision of a Comprehensive Computer Graphics Education in the Context of Computer Games: An Activity-Led Instruction Approach. In *Eurographics 2009 - Education Papers* (2009). doi:10.2312/eged.20091012. 2

[Bus04] BUSS S. R.: Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squares Methods. *IEEE Journal of Robotics and Automation* (2004). 7

[BWF17] BALREIRA D. G., WALTER M., FELLNER D. W.: What we are teaching in Introduction to Computer Graphics. In *Eurographics - Education Papers* (2017). doi:10.2312/eged.20171019. 2

[Cun00] CUNNINGHAM S.: Re-inventing the introductory computer graphics course: providing tools for a wider audience. *Computers & Graphics 24*, 2 (2000), 293–296. doi:10.1016/S0097-8493(99)00164-8. 2

[Ely12] ELYAN E.: Enhanced interactivity and engagement: Learning by doing to simplify mathematical concepts in computer graphics and animation. In *Proc. of the IEEE Global Engineering Education Conference (EDUCON)* (2012), pp. 1–8. doi:10.1109/EDUCON.2012.6201073. 2

[FWW13] FINK H., WEBER T., WIMMER M.: Teaching a modern graphics pipeline using a shader-based software renderer. *Computers & Graphics 37*, 1 (2013). doi:10.1016/j.cag.2012.10.005. 2

[GM98] GLASBEY C. A., MARDIA K. V.: A Review of Image-Warping Methods. *Journal of Applied Statistics 25*, 2 (1998), 155–171. 4

[Kay05] KAY J.: Introduction to Homogeneous Transformations & Robot Kinematics, 2005. Accessed on Jan 19, 2020. URL: http://elvis.rowan.edu/~kay/papers/kinematics.pdf. 6

[LAR*14] LEWIS J. P., ANJYO K., RHEE T., ZHANG M., PIGHIN F. H., DENG Z.: Practice and Theory of Blendshape Facial Models. In *Eurographics 2014 - State of the Art Reports* (2014). doi:10.2312/egst.20141042. 7

[LLWL20] LEE C.-H., LIU Z., WU L., LUO P.: MaskGAN: Towards diverse and interactive facial image manipulation. In *Proc. of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2020), pp. 5549–5558. 6

[Mar98] MARCHESE F. T.: Teaching computer graphics with spreadsheets. In *ACM SIGGRAPH 98 Conference abstracts and applications* (1998), pp. 84–87. 2

[MAT20a] MATLAB, MathWorks. https://es.mathworks.com/products/matlab.html, 2020. 1

[MAT20b] The Origins of MATLAB. https://es.mathworks.com/company/newsletters/articles/the-origins-of-matlab.html, 2020. 1

[MGJ06] MARTÍ E., GIL D., JULIÀ C.: A PBL Experience in the Teaching of Computer Graphics. *Computer Graphics Forum 25*, 1 (2006), 95–103. doi:10.1111/j.1467-8659.2006.00920.x. 2

[PA14] PETERS C. E., ANDERSON E. F.: The Four I's Recipe for Cooking Up Computer Graphics Exercises and Assessments. In *Eurographics – Education Papers* (2014). doi:10.2312/eged.20141029. 2

[SBG10] SCHWEITZER D., BOLENG J., GRAHAM P.: Teaching introductory computer graphics with the processing language. *Journal of Computing Sciences in Colleges 26*, 2 (2010), 73–79. 2

[SM09] SHIRLEY P., MARSCHNER S.: *Fundamentals of Computer Graphics*, 3rd ed. A. K. Peters, Ltd., USA, 2009. 6