Designing a Course on Non-Photorealistic Rendering

Ivaylo Ilinkin[1]

[1]Gettysburg College, USA

Abstract

This paper presents a course design on Non-Photorealistic Rendering (NPAR). As a sub-field of computer graphics, NPAR aims to model artistic media, styles, and techniques that capture salient characteristics in images to convey particular information or mood. The results can be just as inspiring as the photorealistic scenes produced with the latest ray-tracing techniques even though the goals are fundamentally different. The paper offers ideas for developing a full course on NPAR by presenting a series of assignments that cover a wide range of NPAR techniques and shares experience on teaching such a course at the junior/senior undergraduate level.

CCS Concepts

• ***Computing methodologies*** → *Non-photorealistic rendering;*

1. Introduction

Non-Photorealistic Animation and Rendering (NPAR †) is a sub-discipline of *Computer Graphics* that has developed a significant body of work and can be introduced as an independent subject in the curriculum. While traditionally *Computer Graphics* has focused on developing models and algorithms for rendering high-quality realistic-looking images of everyday scenes that are indistinguishable from photographs, NPAR research aims to model artistic media, styles, and techniques that capture salient characteristics in the images, and distort or exaggerate features in scenes to convey particular information or mood. Examples include modeling the physical characteristics of paper and paint to turn a photograph into an image of an oil painting, representing brushstrokes to create an image resembling impressionist painting, simulating pencil drawing to render a technical/medical illustration of 3D objects, automating cartoon and animation rendering, etc. NPAR also has connections with *Image Processing* through algorithms for initial image analysis and decomposition before applying the particular rendering technique.

Despite the accumulated body of work in NPAR in the last 20-25 years and continued active research, NPAR does not appear to have found a place in the curriculum as a stand-alone course. Typically the topic is discussed as a component of a computer graphics course, which does not offer deep exposure to the wide range of ideas generated in NPAR. Some examples of bringing NPAR in the curriculum include systems for demonstrating NPAR algorithms [KNSM07, Mur04], a *Nifty Assignment* for generating ASCII-Art [And17], and a recent paper on teaching image processing for mobile devices through semester-long projects that include implementation of NPAR algorithms [TPD*18].

The First International Symposium on Non-Photorealistic Animation and Rendering in 2000 included a panel on teaching NPAR [SS00]. The panel noted that *"as an emerging area of scientific endeavor, non-photorealistic animation and rendering (NPAR) can potentially take its place in a computer science curriculum"*, but also observed that *"it is a significant challenge to structure in a formal way the area of NPAR around the methods used"*, pointing out that *"the image styles vary greatly within the area, for example from simple-looking black-and-white line drawings to water color animations"*. In other words, the wide range of ideas and techniques makes it difficult to identify a common set that can form the basis for a one-semester course on NPAR. The panel concluded with a commitment to *"look at approaches to structuring the area"*, although it is not clear whether there has been a follow-up meeting to share recent findings.

This paper aims to contribute to the effort launched in [SS00], by suggesting a structure for a one-semester course on NPAR at the junior/senior undergraduate level. The course is built around a collection of papers that introduce various NPAR techniques and are feasible to implement as individual assignments in one to two weeks. The selected papers address the two issues that make it challenging to design a course on NPAR—*breadth* and *feasibility*—while also maintaining some continuity, such that occasionally there are connections between papers across assignments.

The design of the course was guided by the following goals:

† We use the standard abbreviation NPAR, although *animation* was outside the scope of this work.

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

DOI: 10.2312/eged.20201028

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

- expose students to original research
- develop skills for reading a research paper to enable understanding of the main concepts and algorithms
- strengthen programming skills by implementing the proposed algorithms

This was an upper-level elective taught at Gettysburg College, a four-year liberal arts college, as part of the Bachelor's degree in a traditional Computer Science program. The prerequisite for our upper-level courses is *Algorithms and Data Structures* to ensure programming maturity (this is the third course in our program sequence). The students had taken at least one other upper-level course, and that might be a good general recommendation, but prior experience in *Computer Graphics* does not seem essential.

The course had a seminar-style flipped-classroom structure and the students were responsible for conducting a significant portion of the class discussion. The class met twice a week for 75 minutes and each week one paper was assigned as class reading. The students were expected to complete the course assignments individually, but for the class discussion component they were paired randomly each week and each pair worked together to read and understand the main ideas. During the first class meeting one pair led a class discussion and made sure that everyone contributed to the overall understanding of the paper and a possible implementation. To facilitate the discussion, the pairs were asked to (i) share three questions or key observations; (ii) identify all algorithm parameters; and (iii) propose a possible plan for implementation. During the second class meeting the course instructor summarized and clarified the preceding discussion, offered additional material, and discussed the setup for the programming assignment. Occasionally, a paper required an additional day to cover all required concepts.

Note that other arrangements are also possible. For example, an instructor might prefer to have more control over the discussion and use a lecture-style setup. In this case it will still be advisable to assign the papers as readings before class and perhaps gauge the students' effort through a short high-level quiz.

The course used *Python* along with *numpy*, mainly for *vector* and *matrix* operations, and the *Python Imaging Library* (PIL), mainly for drawing lines or points in an image and showing the result. Python was chosen for its simple syntax, flexible constructs, and useful support libraries. This allowed the students to focus on the details of the algorithms and rely on the provided functionality for basic operations. Our department does not teach Python formally, but the students were able to pick it up without much trouble.

2. Course Assignments

In this section we describe the course assignments, which serve as the scaffold for the course. Each assignment introduces a particular NPAR technique—we identify the original research paper that motivated the technique, describe the intended outcome and show results from student work, explain the main ideas behind the algorithm along with pseudocode, and point out connections with previous assignments or discuss implementation details.

The pseudocode is Python-like and an effort is made to balance brevity, precision, and clarity. The hope is that the intended meaning of functions and variables will be clear from context. The goal is to convey a sense of the overall structure of each algorithm and convince the reader that the implementation is feasible. The referenced papers contain the details along with algorithm parameters and their default values. We would also be happy to share additional details and actual implementation with interested instructors.

2.1. Assignment 1: Convolution and Edge Detection

The first assignment served as an introduction to basic concepts in image processing and to programming in Python. The goal was to implement a collection of Python functions modeled after their MATLAB counterparts `conv2` (convolves two matrices), `imfilter` (applies a filter to an image), `edge` (finds edges in an image; computes gradient direction and magnitude), and `fspecial` (creates a pre-defined filter). These are saved in file `utils.py` and used in later assignments.

2.2. Assignment 2: Halftoning

The first NPAR assignment was based on a recursive sub-division technique for producing *halftoning* effect. Halftoning aims to capture the global detail in an image using primitive elements (typically dots) of varying densities or sizes. The algorithm, as described in [Ahm14], is elegantly simple. It recursively subdivides the *input (grayscale) image* into two rectangular sub-regions of roughly equal *density*; when a user-defined *maximum depth* is reached, a dot is placed in the *output image* at the center of the current region. Initially, the whole *input image* is a region R. The density of a region is the sum of the inverted pixel intensities (i.e. *Black* pixels contribute 1 and *White* contribute 0):

```
proc Halftone(inImg, outImg, R, depth):
1   if depth > MAX_DEPTH:
2       outImg[center(R)] = Black //done
3   //split R along longer dimension into
4   //two regions of roughly equal density
5   R1, R2 = densitySplit(inImg, R)
6   halftone(inImg, outImg, R1, depth-1)
7   halftone(inImg, outImg, R2, depth-1)
```

To enhance the halftoning effect [Ahm14] suggests several variations based on line segments instead of points (Figure 1):

- *Edges* – draw the outlines of the final regions
- *Tiles* – draw only the left and bottom edges of the final regions, slightly shortened to avoid connecting adjacent regions
- *Loops* – draw either the vertical or horizontal edges of the final regions (based on split direction) slightly inset; requires some bookkeeping to draw short connecting segments across region boundaries
- *Path* – connect the centers of the final regions with the path that solves the *Traveling Salesperson Problem (TSP)* on the complete graph of the centers using Euclidean distance as the metric; the

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

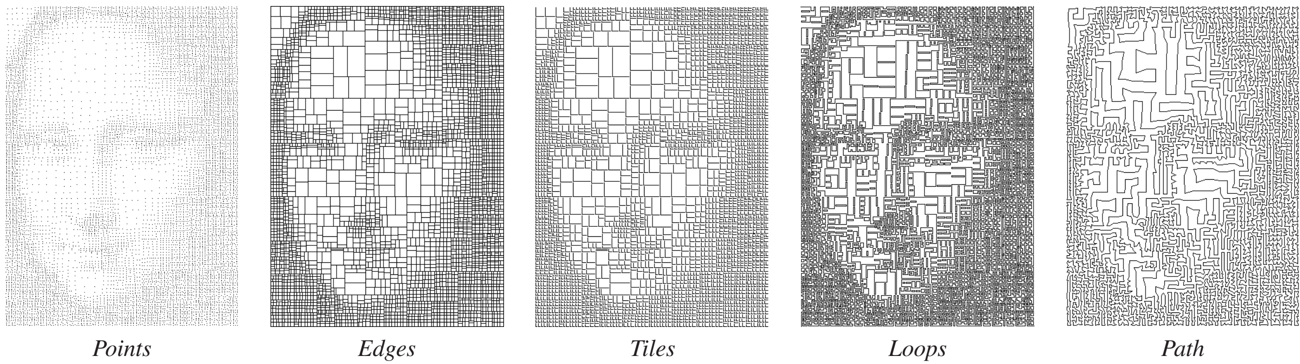

Figure 1: *Assignment 2 results.*

Figure 2: *[Left] Assignment 3* `applyDoG` *computation for single pixel, x, (images from [KLC07]). [Right] Assignment 3 results.*

Concorde TSP solver [ABCC] was used for this step—the students only wrote functions to save/load text files of point coordinates and launched the TSP solver from their Python code as a system command

The algorithm is surprisingly fast even with a naive implementation of computing the density of a sub-region in the image. Nevertheless, the paper suggests two optimizations: precomputing *cumulative density table* and using *binary search* to find the correct place to split.

The application of *binary search* and *TSP* add an extra appeal to the algorithm. It is rare for our students to see a concrete application of *TSP* outside of the *Algorithms and Theory* courses.

2.3. Assignment 3: Line Drawing

Assignment 3 was based on the work in [KLC07] and developed further the concept of gradient and its application to edge detection from Assignment 1. The goal was to produce a *line drawing* by highlighting salient edges in an image (Figure 2). Briefly, the technique in [KLC07] works by constructing an *edge tangent flow* (ETF) which then guides a *difference of Gaussians* (DoG) filter over the image to strengthen and extract the edge information.

```
proc LineDrawing(inImg, outImg):
1    Gx, Gy, G = detectEdges(inImg, Sx, Sy)
2    Tx, Ty = computeTangentField(Gx, Gy, G)
3    Fx, Fy = computeETF(Tx, Ty)
4    H = applyDoG(inImg, Ex, Ey)
5    outImg = threshold(H)
```

Line 1, `detectEdges`, is an application of a *Sobel* operator and has already been completed in Assignment 1. Line 2, rotates the gradient vectors by 90 degrees to compute the tangent field that guides the process of tracing an edge.

Line 3, `computeETF`, is one of the novel contributions of [KLC07] and its purpose is to smooth the tangent field by re-aligning the vectors to ensure coherence in the flow. Each tangent vector, $\vec{t}_p$, at pixel p is updated as a weighted sum of its neighbors, $\vec{t}_q$, in an $n \times n$ radial neighborhood Ω of p as follows:

$$\vec{t}_p \leftarrow \frac{1}{k} \sum_{q \in \Omega(p)} \phi(\vec{t}_p \cdot \vec{t}_q)\, \vec{t}_q\, w_s(p,q)\, w_m(p,q)\, |\vec{t}_p \cdot \vec{t}_q|$$

where:

- ϕ gives the sign of the dot product and its purpose is to align the neighborhood vectors with $\vec{t}_p$
- w_s evaluates to 1 if $||\overline{pq}|| < r$ and to 0, otherwise, so that only neighbors within radius r are considered
- $w_m(p,q) = .5(1 + tanh(\eta(G[q] - G[p])))$ ensures that neighbors with higher gradient are given higher weight; η is a parameter
- the dot product in the last term gives higher weight to neighbors whose direction is more closely aligned with $\vec{t}_p$

Line 4, `applyDoG`, is the second contribution of [KLC07]. The goal of this step is to strengthen the signal for potential edges by following a curve c_x a short distance away from each pixel x guided by the smoothed tangent flow from Line 3. The DoG filter is applied at each step along c_x over a line perpendicular to the tangent

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

vector at the current point and its purpose is to consider neighboring regions for edge flow coherence. The process is illustrated in Figure 2[Left]: For each pixel x we trace a curve along the ETF by applying an iterative step $x \leftarrow x + \alpha \vec{d}_x$, for a user-defined number of iterations, where $\alpha = \pm 1$ and $\vec{d}_x$ is the smoothed gradient direction at x. Next, at each new position x we use the same approach to trace a line perpendicular to the curve using the tangent to the smoothed gradient at x as a fixed direction and compute a difference of two Gaussians with user-defined std. dev. σ_c and σ_s. Essentially this step performs *line integral convolution* where DoG values are collected along c_x.

Line 5, `threshold`, produces the final image by thresholding the result of the DoG filter. In terms of implementation, only Line 4 is a bit more challenging because of the extra bookkeeping involved and the need to understand how to move a point along a curve.

2.4. Assignment 4: Tessellation

This assignment introduced the concept of *particle systems* and their application to creating *irregular tessellations/mosaics* as described in [LM11]. On its own it might have taken an extra week to cover the algorithm, but some components were built in Assignments 2 and 3. This was a nice example of connections across papers and a demonstration of how research builds on previous work.

The goal of this assignment was to subdivide an image into irregular patches, while retaining global information and coherence, so that the result conveys the impression of floor mosaic or stained-glass window. The method works by placing particles on the image that then follow a vector field and trace curves in the output image. A particle stops when it reaches another curve or the boundary. Assuming that the vector field captures features of the image, the particles will trace contours that define regions associated with the features. The regions are then filled with a color that is representative of the pixels in the region (e.g. average color; Figure 3).

```
proc Tessellation(inImg, outImg):
1    particles = generateParticles()
2    Fx, Fy = computeETF(inImg)
3    for p in particles:
5        move p along the ETF
6        record path of p in outImg
7        //stop at boundary or another curve
8    for R in outImg regions:
9        color = averageColor(inImg, R)
10       fillRegion(outImg, R, color)
```

Two questions remain: how to generate the vector field and how to generate the initial locations for the particles. We have answered the first question in Assignment 3 with the computation of ETF, and in fact, this is the suggested method in [LM11]. For the second question, the paper suggests using a regular grid of particles or a method that generates points whose distribution captures salient points in the image, and the latter was computed in Assignment 2.

This means that we already have the tools to complete Lines 1 and 2, and in fact, we also have Lines 3–6 (*move particle along field*), which was a component of `applyDoG` in Assignment 3. In both applications the point/particle moves forward and backward along the field, but whereas in Assignment 3 the field was followed for a fixed number of steps, here the motion continues until a boundary or another curve is reached.

For more abstract effect [LM11] propose replacing the ETF with Lorentz force in a magnetic field. The particle is modeled as having charge q and moving in a magnetic field $\vec{B}$. The force acting on the charged particle is $\vec{F} = q\vec{v} \times \vec{B}$, where the magnetic field is arbitrarily set to $\vec{B} = (0,0,-1)$ and $q(t) = s * (500 - t)^{0.8}$, i.e. the charge varies at each time step t (s is a parameter that controls the shape of the curve). This was a nice illustration of drawing inspiration from another discipline (physics) to model an artistic technique.

We mention briefly the implementation details of Lines 9 and 10. Filling a region in a given color can be done with a simple recursive algorithm that takes as parameters the *output image*, *any pixel* in the region, and the *color*: do nothing if *pixel* is out of bounds or already colored; otherwise, the pixel is set to the given color and the process repeats recursively on its four neighbors. A separate procedure loops through all pixels in the output image and invokes the recursive algorithm on any non-colored pixels (initially the whole image is white with black pixels along the particle curves). This can be adapted for computing the average color in a region.

In terms of implementation, much of the work has been completed in Assignments 2 and 3, namely Lines 1–2. Nevertheless, we gave the students precomputed matrices (saved as `numpy` objects) with initial particle placements and ETF in order to eliminate unnecessary re-computation of the same information and to enable progress in case of issues with the code in previous assignments.

2.5. Assignment 5: Voronoi Mosaics

Assignment 5 also looked at the process of generating mosaics, but used a substantially different approach from that of Assignment 4. It was based on the work in [Sec02] which actually introduces a method for producing *stippling* effect, but we modified the algorithm slightly to generate tilings that appear to be built out of regular hexagons. The input in our case is a color image, although much of the processing takes place in its grayscale version.

This assignment introduced the concept of *Voronoi Diagram* from *Computational Geometry*, a topic that our students usually do not encounter in the curriculum. In 2D the *Voronoi Diagram* of a set of n points/sites is a partitioning of the plane into n regions such that region R_i contains all points that are closest to site p_i. A *Centroidal Voronoi Diagram* (CVD) is a *Voronoi Diagram* such that each site is also the center of mass of its corresponding region given a density function $\rho(x,y)$.

The main idea behind the algorithm in [Sec02] is to compute a CVD for an initial set of randomly placed points inside the grayscale version of the given color image. The density function is $\rho(i,j) = 1 - intensity(i,j)$ (i.e. *Black* pixels contribute 1 and *White* contribute 0). There is an obvious connection with Assignment 2, which also used the same density function for *halftoning* effect.

The CVD can be computed using an iterative process known

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

Figure 3: *Assignment 4 results: original (left), ETF-based (middle), Lorentz field (right).*

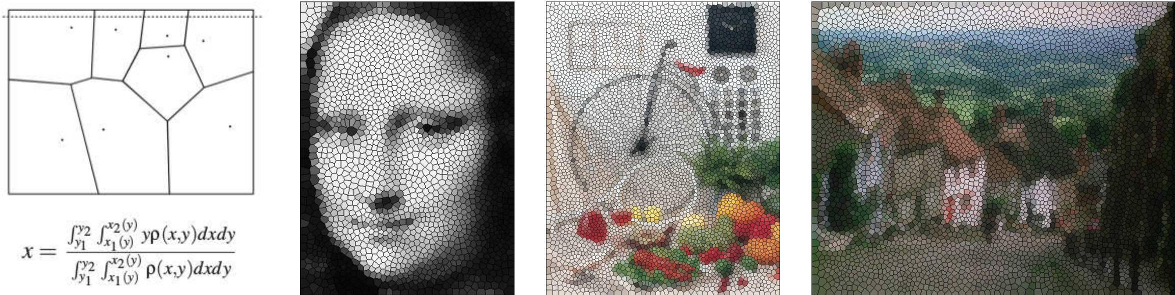

Figure 4: *[Left] Centroid Computation. [Right] Assignment 5 results.*

Figure 5: *Assignment 6 results.*

as *Lloyd's method* [DFG99] which alternates between computing a *Voronoi Diagram* for the current set of points and then moving/replacing the points with the centroids of their corresponding regions. The process stops when a convergence criterion is met (for example, *average distance moved* below a user-defined threshold).

```
proc VoronoiMosaic(inImg, outImg):
1   points = generatePoints()
2   while not converged:
3       CVD = computeVoronoi(points)
4       for R in regions(CVD):
5           computeCentroid(inImg, R)
6       points = region centroids
7   for R in regions(CVD):
8       color = averageColor(inImg, R)
9       fillRegion(outImg, R, color)
```

The sites of the final CVD are used as stipple points, based on the idea that "denser" regions will attract more sites. In our modification, we instead render the regions of the CVD overlaid on the original color image. Each region is assigned the average color of the enclosed pixels, which gives the mosaic effect (Figure 4).

The students were given a program for computing the *Voronoi Diagram* in Line 3. The output of the program was a list of edges along with information about the index/id of the region to the left and right of each edge. The challenge for the students was to efficiently process all regions and compute their centroids in Lines 4–5. Figure 4[Left] shows the expression for computing the x coordinate of a centroid, where the denominator is the region density and the numerator is the moment about the y-axis. The students were asked to implement a *sweep-line* algorithm that scans the image one row at a time and at each step considers the intersected *Voronoi* edges sorted by x coordinate (Figure 4[Left]). The pixel densities along each scan segment are added to the current density total of

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

the corresponding region. The moment about the y-axis for each region can be computed similarly along the way. When the sweep line leaves the image, the x-coordinates of all centroids can be computed. Similar discussion applies to the y coordinate. Finally, note that Lines 7–9 are just Lines 8–10 in Assignment 4.

2.6. Assignment 6: Impressionist Effect

Assignment 6 revisited the concept of *gradient* and its application to guiding brush strokes for *impressionist* effect as described in [Lit97]. The main idea behind the algorithm is to draw short linear strokes distributed randomly across the image using the gradient to determine the direction of each stroke.

```
proc Impressionist(inImg, outImg):
1   inImg' = smoothImage(inImg, Gauss)
2   Gx, Gy, G = detectEdges(inImg', Sx, Sy)
3   centers = generateGrid(inImg)
4   for c in shuffle(centers):
5       p1, p2 = followGradient(Gx, Gy, c)
6       color = inImg[c]
7       drawStroke(outImg, p1, p2, color)
```

The algorithm begins in Lines 1 and 2 with an application of a *Sobel operator* over a Gaussian filtered input image. Line 3 selects stroke centers—the suggestion in [Lit97] is to place the stroke centers on a regular grid with a user-defined initial spacing (half stroke width, for example); however, to prevent artifacts, the strokes are rendered in random order by shuffling the generating centers. Finally, the strokes are rendered in Lines 4–7: First the endpoints $(p1, p2)$ of a stroke are found by following perpendicular to the gradient for user-defined half stroke length in both directions starting at the stroke center, $p = c$, as discussed in Assignment 3 and Figure 2 (Left). Next, the endpoints $(p1, p2)$ are connected with an antialiased line of user-defined width (plus a small random perturbation) in the color of the generating center in the original image.

The students noted the connection with Assignment 3 (curve following) and Assignment 1 (*image smoothing* and *edge detection* for clipping strokes against identifiable objects in the image). We also had an opportunity to discuss the concept of *antialiasing*; however, in order to keep the focus on the rendering technique, in the implementation the antialiased stroke lines were render using PIL.

2.7. Assignment 7: Oil Painting Effect

Assignment 7 was based on the work described in [Her02]. The goal was to produce a rendering of an image that conveys an *oil painting* effect, and in particular, an *impasto effect*, which is characterized by the layering of thick paint strokes that can produce three dimensional relief texture over the surface of the painting.

The algorithm described in [Her02] can be used as a post-processing step to any stroke-based technique. The main idea is to create a height field by applying the strokes over the image and accumulating *paint* over the areas/pixels covered by each stroke. Viewed as a quadrilateral surface mesh over the pixel grid, the height field can be used to compute surface normals at each pixel/vertex from the normals of the incident quads. Finally, the actual color image produced by the original stroke-based technique is rendered using bump-mapping with the Phong shading model, which creates the appearance of relief texture over the image.

The application of the strokes in the construction of the height field is guided by a *height map*, which controls the variation in stroke thickness. The stroke is subdivided into a grid and *uv*-mapped to the *height map* to determine the height at each point, which is then composited with the height field (Figure 7). An *opacity map* can also be used as the strokes are applied and composited for the final rendering.

The user can create several *height*/*opacity* map pairs and associate them randomly with the strokes, although compelling results can be produced with a single pair, which is the approach used in [Her02] and the approach that we follow. The *height* and *opacity* maps are simple to create—the ones shown in Figure 7 are created by the author of [Her02] in a *Paint* program as random pattern.

```
proc OilPainting(inImg, strokes, maps):
1   hField = makeMatrix(size(inImg)
2   for s in strokes:
3       hMap, oMap = randomEntry(maps)
4       for u,v in [0,1]x[0,1]:
5           p = s[u, v]
6           h = htMap[u, v]
7           a = opMap[u, v]
8           hField[p] = a*hMap[u, v] +
9                       (1-a)*hField[p]
10   normals = makeMatrix(size(inImg))
11   for r,c in size(hField):
12       normals = norm(N1+N2+N3+N4)
13   //use bump mapping and Phong model
14   //to render inImg with normals
```

Line 1 creates a scalar matrix that stores the height at each pixel. Lines 2–3 process the strokes one at a time with randomly selected *height map*/*opacity map* pair per stroke. Lines 4–9 apply the stroke by defining a [0,1]x[0,1] (u, v) grid over the stroke and sampling at user-defined steps the height value, h, together with a compositing factor, a, from the *height map* and *opacity map*, respectively, in a process analogous to texture mapping. Line 10 creates a vector matrix for the normals at each pixel/vertex and Lines 11–13 compute each normal from the normals of the pixel's/vertex's four adjacent quads in the *height field* surface mesh.

This assignment was a natural extension of Assignment 6. We used the stroke sequences from Assignment 6 and rendered the images again, but now they had relief textured appearance. This was also an opportunity to discuss important concepts in computer graphics, namely *Phong illumination model*, *texture mapping*, and *bump mapping*. The students were particularly intrigued by the possibility to produce different visual effects simply by changing the surface normals without altering the underlying geometry.

We also used this as an opportunity to give a high-level intro-

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

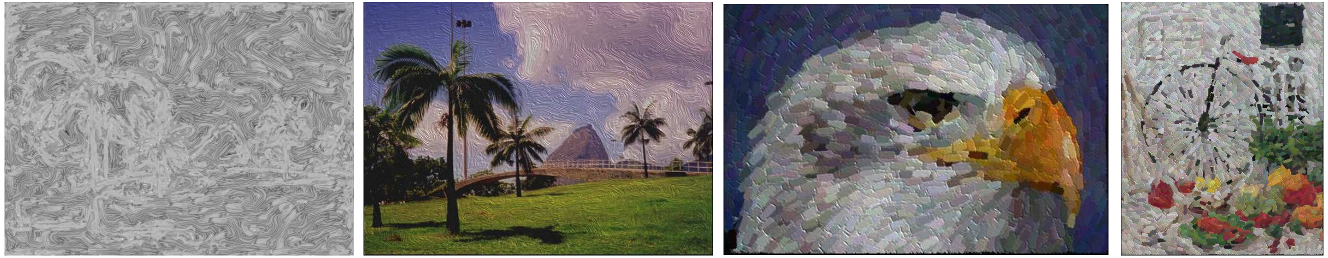

Figure 6: *Assignment 7 results.*

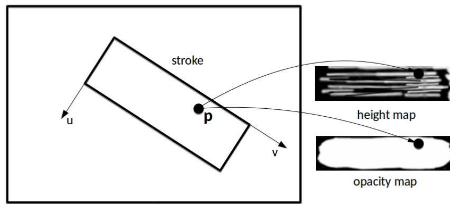

Figure 7: *Stroke and height/opacity maps correspondence.*

duction to OpenGL. The students were given a simple renderer for drawing a collection of points given the color and normal vector at each point and we discussed the various steps in setting up the renderer. Our goal was to offer only basic familiarity with the main ideas—the students did not write OpenGL code, but simply wrote to a file a sequence of *point/color/normal* specifications that were read and displayed by the renderer.

2.8. Assignment 8: Pixel Art

Assignment 8 introduced a technique for generating reduced color palettes in order to represent an image in the style of *pixel art* [GDA*12]. In other words, the goal is to select a limited number of colors and assign them to the individual pixels to best convey the original impression (Figure 8). The user specifies the number of desired colors in the final palette and the dimensions of the output image. The algorithm builds the palette in an iterative *deterministic annealing* clustering process [Ros98]. The deterministic annealing step is interleaved with refinement of *superpixels* [ASS*10], which represent regions in the image of original pixels that are clustered based on minimizing the following distance metric:

$$d(p_i, p_s) = d_c(p_i, p_s) + m\sqrt{\frac{N}{M}} d_p(p_i, p_s) \qquad (1)$$

where p_i is a pixel in the image, p_s is a superpixel, d_c is the Euclidean distance in LAB color space and d_p is the Euclidean distance in pixel coordinates, and m is a parameter; N and M represent the total number of pixels in the output and input image, respectively. In other words a *superpixel* represents a region in the input image of pixels that are close in proximity and color. The color of the superpixel is eventually assigned in the output image to represent the whole region from the input.

```
proc PixelArt(inImg, outW, outH, K):
1    superpix = buildGrid(inImg, outW, outH)
2    palette = [ meanColor(inImg) ]
3    T = 1.1Tc
4    while T > Tf
5        refine(superpix)
6        associate(superpix, palette)
7        refine(palette)
8        if palette converged:
9            T = αT
10           expand(palette)
11   //build output image:
12   // assign to each output pixel palette
13   // color of corresponding superpixel
```

The algorithm from [GDA*12] takes as input the *original image*, the *dimensions* of the output image, and the *palette size* for the output image. In Line 1 the superpixels are selected as the centers of a `outW` x `outH` grid over the input image, so that initially each superpixel represents a rectangular region of pixels in the original image. On each iteration this association is improved in Line 5 by recomputing Equation 1 for each pixel against each superpixel.

The palette is initialized to a single color and all superpixels are associated with that color. The ultimate goal is to split the superpixels int K clusters, which is achieved through the annealing process, which essentially performs k-clustering. During the annealing process the clusters split along the principal directions generating new colors in the palette; the details are described in [Ros98].

This assignment required reading 3 different papers and took a bit longer to complete, but overall the implementation was feasible.

3. Discussion

This paper introduced a design for a course in *Non-Photorealistic Rendering*. The course is built around a selection of research papers that cover a wide range of techniques and are feasible to implement in one to two weeks at the junior/senior undergraduate level. In addition, there were strong connections between some of the papers, which helped integrate the material and gave it a cohesive flow.

In addition to introducing the students to the exciting field of NPAR the course also aimed to develop skills in reading published

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

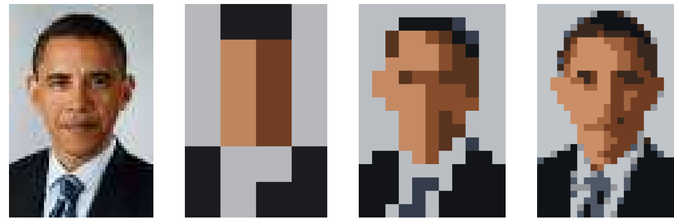

Figure 8: *Assignment 8 results—original image 54x80 / 4x6,4 colors / 11x16, 6 colors / 22x32, 8 colors.*

research and strengthen programming skills. The course feedback suggests that these goals were largely met. The students described the course as *hard but fun* and noted as positive aspects:

- *learned how to take a scientific paper and implement it myself*
- *ability to read, understand and implement journal articles*
- *debugging ability, patience*
- *feel very confident in Python now and scientific papers seem much more feasible now*
- *skill to read paper and grab important information from paper*
- *time management, debugging skills; learning how to write clean code help*
- *combined knowledge from other areas (physics, math, etc)*
- *presentation skills*

This was the first offering of the course, so there were challenges along the way. In particular, the students expressed frustration with the difficulty of debugging, which unfortunately is a challenge in any computer graphics project. The fact that several students shared that they developed debugging skills hopefully suggests that the difficulty leveled off as the course progressed. Whenever possible we offered precomputed data from a previous assignment to reduce idle time and ensure progress if there were problems with previous code. Finally, there were occasionally times of delay when a particular step in an algorithm was unclear and adjustments were made in the course schedule to account for this. Fortunately, this did not happen often and the students showed understanding. A future offering will benefit from the experience gained this semester.

In the future, we would like to explore the possibility of including a paper on ASCII-Art. One of the papers in the course was on *image inpainting* (not discussed here), so this could be replaced if a feasible ASCII-Art option is found (one candidate is [And17]).

References

[ABCC] APPLEGATE D., BIXBY R., CHVÁTAL V., COOK W.: Concorde TSP Solver. http://www.math.uwaterloo.ca/tsp/concorde/. 3

[Ahm14] AHMED A.: Modular Line-based Halftoning via Recursive Division. In *Proceedings of the Workshop on Non-Photorealistic Animation and Rendering* (New York, NY, USA, 2014), NPAR '14, ACM, pp. 41–48. doi:10.1145/2630397.2630403. 2

[And17] ANDERSON E.: Generating ASCII-Art: A Nifty Assignment from a Computer Graphics Programming Course. In *EG 2017 - Education Papers* (2017), Bourdin J.-J., A. S., (Eds.), The Eurographics Association. doi:10.2312/eged.20171021. 1, 8

[ASS*10] ACHANTA R., SHAJI A., SMITH K., LUCCHI A., FUA P., SÜSSTRUNK S.: *SLIC Superpixels*. Tech. rep., EPFL Technical Report 149300, June 2010. 7

[DFG99] DU Q., FABER V., GUNZBURGER M.: Centroidal voronoi tessellations: Applications and algorithms. *SIAM Rev. 41*, 4 (Dec. 1999), 637–676. doi:10.1137/S0036144599352836. 5

[GDA*12] GERSTNER T., DECARLO D., ALEXA M., FINKELSTEIN A., GINGOLD Y., NEALEN A.: Pixelated Image Abstraction. In *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering* (Goslar Germany, Germany, 2012), NPAR '12, Eurographics Association, pp. 29–36. URL: http://dl.acm.org/citation.cfm?id=2330147.2330154. 7

[Her02] HERTZMANN A.: Fast Paint Texture. In *Proceedings of the 2Nd International Symposium on Non-photorealistic Animation and Rendering* (New York, NY, USA, 2002), NPAR '02, ACM, pp. 91–ff. doi:10.1145/508530.508546. 6

[KLC07] KANG H., LEE S., CHUI C.: Coherent Line Drawing. In *Proceedings of the 5th International Symposium on Non-photorealistic Animation and Rendering* (New York, NY, USA, 2007), NPAR '07, ACM, pp. 43–50. doi:10.1145/1274871.1274878. 3

[KNSM07] KONDO K., NISHITA T., SATO H., MATSUDA K.: An Educational Non-Photorealistic Rendering System Using 2D Images by Java Programming. *Journal for Geometry and Graphics 11*, 2 (Jan 2007), 237–247. 1

[Lit97] LITWINOWICZ P.: Processing Images and Video for an Impressionist Effect. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., pp. 407–414. doi:10.1145/258734.258893. 6

[LM11] LI H., MOULD D.: Artistic Tessellations by Growing Curves. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Non-Photorealistic Animation and Rendering* (New York, NY, USA, 2011), NPAR '11, ACM, pp. 125–134. doi:10.1145/2024676.2024697. 4

[Mur04] MURMAN C.: Teaching Tool to Demonstrate Techniques Used in Non-Photorealistic Rendering, 2003–2004. BSc Thesis. 1

[Ros98] ROSE J.: Deterministic annealing for clustering, compression, classification, regression, and related optimization problems. *Proceedings of the IEEE 86*, 11 (Nov 1998), 2210–2239. doi:10.1109/5.726788. 7

[Sec02] SECORD A.: Weighted Voronoi Stippling. In *Proceedings of the 2nd International Symposium on Non-photorealistic Animation and Rendering* (New York, NY, USA, 2002), NPAR '02, ACM, pp. 37–43. doi:10.1145/508530.508537. 4

[SS00] STROTHOTTE T., SCHLECHTWEG S.: Teaching Non-photorealistic Animation and Rendering (Panel Session). In *Proceedings of the 1st International Symposium on Non-photorealistic Animation and Rendering* (New York, NY, USA, 2000), NPAR '00, ACM, p. 109. doi:10.1145/340916.340932. 1

[TPD*18] TRAPP M., PASEWALDT S., DÜRSCHMID T., SEMMO A., DÖ J.: Teaching Image-Processing Programming for Mobile Devices: A Software Development Perspective. In *EG 2018 - Education Papers* (2018), Post F., Žára J., (Eds.), The Eurographics Association. doi:10.2312/eged.20181002. 1

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.