Learning For CG

Author: Caoliwen Wang

Institute: School of Mathematical Sciences USTC

Date: July 24, 2024

Version: 4.3

Contents

Chapter 1 Numerical Optimization **1**

1.1 Introduction . . . 1

1.2 Fundamentals of Unconstrained Optimization . . . 1

1.2.1 What Is A Solution? . . . 1

1.2.2 Algorithm Overview . . . 1

Chapter 2 Computer Graphics Course in USTC **2**

2.1 Image Warping . . . 2

2.1.1 Problem Statement . . . 2

2.1.2 Propose Algorithms . . . 2

2.1.2.1 IDW . . . 2

2.1.2.2 RBF . . . 3

2.1.2.3 MLS . . . 4

2.2 Poisson Editing . . . 7

2.2.1 Problem Statement . . . 7

2.2.2 Propose Algorithms . . . 8

2.3 Tutte Parameterization . . . 8

2.3.1 Problem Statement . . . 8

2.3.2 Propose Algorithms . . . 9

2.3.3 Programming . . . 11

2.3.3.1 Halfedge Structure . . . 11

2.3.3.2 Node programming . . . 11

2.4 ARAP Parameterization . . . 12

2.4.1 Problem Statement . . . 12

2.4.2 Propose Algorithms . . . 12

2.5 Phong Shading . . . 14

2.5.1 Problem Statement . . . 14

2.5.2 Propose Algorithms . . . 15

2.5.2.1 Blinn-Phong Shading Model . . . 15

2.5.2.2 Shadow Mapping . . . 15

2.5.2.3 Normal Mapping . . . 15

2.6 Path Tracing . . . 16

2.6.1 Problem Statement . . . 16

2.6.2 Propose Algorithms . . . 16

2.6.2.1 Physics Based Rendering . . . 16

2.6.2.2 Ray Tracing . . . 17

2.6.2.3 Monte Carlo sampling method . . . 19

2.6.2.4 Pace Tracing 20
2.7 Mass Spring 22
2.7.1 Problem Statement 22
2.7.2 Propose Algorithms 22
2.7.2.1 Semi-Implicit Method 22
2.7.2.2 Implicit Euler Integration 23
2.7.2.3 Interactions 26
2.8 SPH Fluid 26
2.8.1 Problem Statement 26
2.8.2 Propose Algorithms 27
2.8.2.1 Navier-Stokes Equations 27
2.8.2.2 Spatiotemporal Discretization 28
2.8.2.3 Computational Formula 29
2.9 Character Animation 30
2.9.1 Problem Statement 30
2.9.2 Propose Algorithms 31
2.9.2.1 Skeletal Animation 31
2.9.2.2 Skinning Animation 32

Chapter 3 Modeling Course in USTC **33**
3.1 Seam Carving for Content-Aware Image Resizing 33
3.1.1 Problem Statement 33
3.1.2 Propose Algorithms 33
3.1.2.1 The Energy Function 33
3.1.2.2 Seams 33
3.1.2.3 Optimization 34
3.1.2.4 Resizing 34
3.2 Image Processing 34
3.2.1 Problem Statement 34
3.2.2 Propose Algorithms 35
3.2.2.1 Image Compression 35
3.2.2.2 Low-rank Image Restoration 37
3.3 Interpolation and Neural Work 38
3.3.1 Problem Statement 38
3.3.2 Propose Algorithms 38
3.3.2.1 Curve Reconstruction 38
3.3.2.2 Insect Classification Models 40

Chapter 4 GAMES102: Geometry Modeling And Processing **43**
4.1 Data Fitting 43

Preface

Chapter 1 Numerical Optimization

This Chapter comes from 《Numerical Optimization》, Jorge Nocedal; Stephen J. Wright.

1.1 Introduction

First, we need understand the process of modeling and some crucial concepts.

1. objective
2. variable
3. constrained
4. optimality conditions

So the modeling means that we need to maximize or minimize the objective function.
Additionally, we need to classify the modeling problems in many ways:

1. Linear or nonlinear?
2. discrete or continuous?
3. constrained or unconstrained?
4. Global or local?
5. stochastic or deterministic?
6. convexity

Finally, we need to think about the robustness, efficiency and accuracy, although sometimes they might contradict to each other.

1.2 Fundamentals of Unconstrained Optimization

Fortunately, our algorithms get to choose these points, and they try to do so in a way that identifies a solution reliably and without using too much computer time or storage. Often, the information about f does not come cheaply, so we usually prefer algorithms that do not call for this information unnecessarily.

1.2.1 What Is A Solution?

First, some concepts of local minimum and global minimum.

Then, here are some theorems: P14-P17. These results, which are based on elementary calculus, provide the foundations for unconstrained optimization algorithms.

In the end, there exist interesting problems that own bad regularity.

1.2.2 Algorithm Overview

Chapter 2 Computer Graphics Course in USTC

This chapter comes from Computer Graphics Course in USTC, 2024, taught by Ligang Liu.

The homepage is listed: http://staff.ustc.edu.cn/~lgliu/Courses/ComputerGraphics_2024_spring-summer/default.htm

2.1 Image Warping

2.1.1 Problem Statement

We need to implement image warping. We have implemented interactive mouse dragging, allowing users to designate certain points as fixed and move others to desired positions. These selected points are collectively referred to as control points$(\boldsymbol{p}_i, \boldsymbol{q}_i)$, $\boldsymbol{p}_i, \boldsymbol{q}_i \in \mathbb{R}^2$, $i = 1, \ldots, n$. Taking these control points as input, our goal is to output the deformed entire image. Essentially, the problem entails determining a distortion function $\boldsymbol{f}$ given n interpolation conditions:

$$\boldsymbol{f}(\boldsymbol{p}_i) = \boldsymbol{q}_i, i = 1, \ldots, n$$

We then apply $\boldsymbol{f}$ to all points in the image to determine the positions of the deformed points. Finally, we assign the pixel values of the points before distortion to the corresponding points after distortion, achieving the rendering of the deformed image.

2.1.2 Propose Algorithms

The essence of this problem lies in finding the interpolation function. Here, we present three different approaches:

1. IDW: Inverse distance-weighted interpolation methods;
2. RBF: Radial basis functions interpolation method;
3. MLS: Moving least squares interpolation method.

I need to clarify that the Moving Least Squares (MLS) method is an assignment in a mathematical experiment course. Here, it has been implemented once again using C++.

2.1.2.1 IDW

We need to define some local interpolation functions $\boldsymbol{f}_i(\boldsymbol{p}) : \mathbb{R}^2 \to \mathbb{R}^2$, which satisfy the following conditions:

$$\boldsymbol{f}_i(\boldsymbol{p}_i) = \boldsymbol{q}_i.$$

To be more specific,

$$\boldsymbol{f}_i(\boldsymbol{p}) = \boldsymbol{q}_i + \boldsymbol{D}_i(\boldsymbol{p} - \boldsymbol{q}_i). \tag{2.1.1}$$

In equation (2.1.1), $\boldsymbol{D}_i : \mathbb{R}^2 \to \mathbb{R}^2$, which satisfies that $\boldsymbol{D}_i(\mathbf{0}) = \mathbf{0}$.

Choose $\boldsymbol{D}_i$ to be a linear transformation. We can define it in a more accurate way. Define the energy function:

$$E_i(\boldsymbol{D}_i) = \sum_{j=1, j\neq i}^{n} \sigma_i(\boldsymbol{p}_j) \|\boldsymbol{q}_i + \boldsymbol{D}_i(\boldsymbol{p}_j - \boldsymbol{p}_i) - \boldsymbol{q}_j\|^2. \tag{2.1.2}$$

Let the $\boldsymbol{D_i}$ minimize the energy function.

$$\frac{\partial E_i}{\partial \boldsymbol{D}_i} = \sum_{j=1, j\neq i}^{n} \sigma_i(\boldsymbol{p}_j)[\boldsymbol{q}_i + \boldsymbol{D}_i(\boldsymbol{p}_j - \boldsymbol{p}_i) - \boldsymbol{q}_j] \cdot (\boldsymbol{p}_j - \boldsymbol{p}_i)^\top = 0. \tag{2.1.3}$$

So $\boldsymbol{D_i}$ can be calculated in the following way.

$$[\sum_{j=1, j\neq i}^{n} \sigma_i(\boldsymbol{p}_j)(\boldsymbol{q}_j - \boldsymbol{q}_i)(\boldsymbol{p}_j - \boldsymbol{p}_i)^\top] \cdot [\sum_{j=1, j\neq i}^{n} \sigma_i(\boldsymbol{p}_j)(\boldsymbol{p}_j - \boldsymbol{p}_i)(\boldsymbol{p}_j - \boldsymbol{p}_i)^\top]^{-1} \tag{2.1.4}$$

We use the Eigen library to compute the inverse of a matrix.

The complete interpolation function is

$$\boldsymbol{f}(\boldsymbol{x}) = \sum_{i=1}^{n} \omega_i(\boldsymbol{x}) \boldsymbol{f}_i(\boldsymbol{x}) \tag{2.1.5}$$

$\omega_i : \mathbb{R}^2 \to \mathbb{R}$ is defined:

$$\omega_i(\boldsymbol{x}) = \frac{\sigma_i(\boldsymbol{x})}{\sum_{j=1}^{n} \sigma_j(\boldsymbol{p})}. \tag{2.1.6}$$

$$\sigma_i(\boldsymbol{x}) = \frac{1}{\|\boldsymbol{x} - \boldsymbol{x}_i\|^\mu} \tag{2.1.7}$$

μ is a constant.

2.1.2.2 RBF

Assuming the interpolation function f is in the form of a combination of radial basis functions as follows:

$$f(\boldsymbol{p}) = \sum_{i=1}^{n} \boldsymbol{\alpha}_i R(\|\boldsymbol{p} - \boldsymbol{p}_i\|) + \boldsymbol{A}\boldsymbol{p} + \boldsymbol{b}. \tag{2.1.8}$$

In equation (2.1.8),

1. $R(\|\boldsymbol{p} - \boldsymbol{p}_i\|)$ consists of n radial basis functions, such as the option $R(d) = (d^2 + r^2)^{\mu/2}$, where the coefficients $\boldsymbol{\alpha}_i \in \mathbb{R}^2$ are to be determined;
2. $\boldsymbol{A} \in \mathbb{R}^{2\times 2}$ and $\boldsymbol{b} \in \mathbb{R}^2$ are the to-be-determined affine parts.

The mapping has $2(n+3)$ degrees of freedom, and the interpolation conditions are:

$$f(\boldsymbol{p}_j) = \sum_{i=1}^{n} \boldsymbol{\alpha}_i R(\|\boldsymbol{p}_j - \boldsymbol{p}_i\|) + A\boldsymbol{p}_j + \boldsymbol{b} = \boldsymbol{q}_j, \quad j = 1, \ldots, n. \tag{2.1.9}$$

Provided are $2n$ constraints. Optional additional constraints include:

$$\begin{pmatrix} \boldsymbol{p}_1 & \cdots & \boldsymbol{p}_n \\ 1 & \cdots & 1 \end{pmatrix}_{3\times n} \begin{pmatrix} \boldsymbol{\alpha}_1^\top \\ \vdots \\ \boldsymbol{\alpha}_n^\top \end{pmatrix}_{n\times 2} = \boldsymbol{0}_{3\times 2}. \tag{2.1.10}$$

Based on the above constraints, we can formulate the following matrix equation to solve for the parameters to be determined:

$$\begin{pmatrix}
R(\|\boldsymbol{p_1}-\boldsymbol{p}_1\|) & \cdots & R(\|\boldsymbol{p_1}-\boldsymbol{p}_n\|) & 0 & \cdots & 0 & \boldsymbol{p}_1[0] & \boldsymbol{p}_1[1] & 0 & 0 & 1 & 0 \\
R(\|\boldsymbol{p_2}-\boldsymbol{p}_1\|) & \cdots & R(\|\boldsymbol{p_2}-\boldsymbol{p}_n\|) & 0 & \cdots & 0 & \boldsymbol{p}_2[0] & \boldsymbol{p}_2[1] & 0 & 0 & 1 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
R(\|\boldsymbol{p_n}-\boldsymbol{p}_1\|) & \cdots & R(\|\boldsymbol{p_n}-\boldsymbol{p}_n\|) & 0 & \cdots & 0 & \boldsymbol{p}_n[0] & \boldsymbol{p}_n[1] & 0 & 0 & 1 & 0 \\
0 & \cdots & 0 & R(\|\boldsymbol{p_1}-\boldsymbol{p}_1\|) & \cdots & R(\|\boldsymbol{p_1}-\boldsymbol{p}_n\|) & 0 & 0 & \boldsymbol{p}_1[0] & \boldsymbol{p}_1[1] & 0 & 1 \\
0 & \cdots & 0 & R(\|\boldsymbol{p_2}-\boldsymbol{p}_1\|) & \cdots & R(\|\boldsymbol{p_2}-\boldsymbol{p}_n\|) & 0 & 0 & \boldsymbol{p}_2[0] & \boldsymbol{p}_2[1] & 0 & 1 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & \cdots & 0 & R(\|\boldsymbol{p_n}-\boldsymbol{p}_1\|) & \cdots & R(\|\boldsymbol{p_n}-\boldsymbol{p}_n\|) & 0 & 0 & \boldsymbol{p}_n[0] & \boldsymbol{p}_n[1] & 0 & 1 \\
\boldsymbol{p}_1[0] & \cdots & \boldsymbol{p}_n[0] & 0 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 0 \\
\boldsymbol{p}_1[1] & \cdots & \boldsymbol{p}_n[1] & 0 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 0 \\
1 & \cdots & 1 & 0 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 0 \\
0 & \cdots & 0 & \boldsymbol{p}_1[0] & \cdots & \boldsymbol{p}_n[0] & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\
0 & \cdots & 0 & \boldsymbol{p}_1[1] & \cdots & \boldsymbol{p}_n[1] & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\
0 & \cdots & 0 & 1 & \cdots & 1 & 0 & \cdots & \cdots & \cdots & \cdots & 0
\end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{\alpha}_1[0] \\ \vdots \\ \boldsymbol{\alpha}_n[0] \\ \boldsymbol{\alpha}_1[1] \\ \vdots \\ \boldsymbol{\alpha}_n[1] \\ a_{11} \\ a_{12} \\ a_{21} \\ a_{22} \\ b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} \boldsymbol{q}_1[0] \\ \vdots \\ \boldsymbol{q}_n[0] \\ \boldsymbol{q}_1[1] \\ \vdots \\ \boldsymbol{q}_n[1] \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{2.1.11}$$

In equation (2.1.11),

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix},$$

$\boldsymbol{p}_i[0]$ represents the x-coordinate of point $\boldsymbol{p}_i$, $\boldsymbol{p}_i[1]$ represents the y-coordinate of point $\boldsymbol{p}_i$, as well as $\boldsymbol{q}_i[0]$, $\boldsymbol{q}_i[1]$, $\boldsymbol{\alpha}_i[0]$, $\boldsymbol{\alpha}_i[1]$.

We solve the expression (2.1.11) using the Eigen library and return the solution vector, which contains the coefficients to be determined.

2.1.2.3 MLS

In this project, I just finished the rigid transformation, but complete processes mentioned in the paper are listed.

The known deformation points are denoted as $\boldsymbol{p}$, and their images are denoted as $\boldsymbol{q}$, stored in $n \times 2$ matrices $\boldsymbol{p}$ and $\boldsymbol{q}$ respectively, where n represents the number of fixed points. Based on the given $\boldsymbol{p}$ and $\boldsymbol{q}$. We define:

$$l_v(\boldsymbol{x}) = \boldsymbol{x}\boldsymbol{M} + \boldsymbol{T}. \tag{2.1.12}$$

The transformation needs to minimize the following energy function:

$$E = \sum_i \omega_i |l_v(\boldsymbol{p}_i) - \boldsymbol{q}_i|^2. \tag{2.1.13}$$

attains its minimum value, where ω_i represents the weight vector, a $1 \times n$ vector defined as:

$$\omega_i = \frac{1}{|\boldsymbol{p}_i - \boldsymbol{v}|^{2\alpha}}. \tag{2.1.14}$$

From here, it can be seen that this weight vector depends on $\boldsymbol{v}$ and $\boldsymbol{p}$. Substituting the assumption equation (2.1.12) into the energy function (2.1.13), we can obtain:

$$E = \sum_i w_i \left|\boldsymbol{p}_i M + \boldsymbol{T} - \boldsymbol{q}_i\right|^2 . \tag{2.1.15}$$

Taking the derivative of Equation (2.1.15) with respect to $\boldsymbol{T}$, and since the energy function attains an extremum, we have:

$$\frac{\partial E}{\partial \boldsymbol{T}} = 2\sum_i \omega_i \left|\boldsymbol{p}_i \boldsymbol{M} + \boldsymbol{T} - \boldsymbol{q}_i\right| = 0. \tag{2.1.16}$$

Then

$$\boldsymbol{T} = \boldsymbol{q}^* - \boldsymbol{p}^* \boldsymbol{M}. \tag{2.1.17}$$

We define $\boldsymbol{q}^*$ and $\boldsymbol{p}^*$:

$$\begin{aligned} \boldsymbol{p}_* &= \frac{\sum_i w_i \boldsymbol{p}_i}{\sum_i w_i}. \\ \boldsymbol{q}_* &= \frac{\sum_i w_i \boldsymbol{q}_i}{\sum_i w_i}. \end{aligned} \tag{2.1.18}$$

We can find that $\boldsymbol{p}^*$ and $\boldsymbol{q}^*$ are only up to ω, $\boldsymbol{p}$ and $\boldsymbol{q}$. Then

$$l_v(\boldsymbol{x}) = (\boldsymbol{x} - \boldsymbol{p}^*)\boldsymbol{M} + \boldsymbol{q}^*. \tag{2.1.19}$$

At this point, we can rewrite the energy function as:

$$E = \sum_i w_i \left|\hat{\boldsymbol{p}}_i \boldsymbol{M} - \hat{\boldsymbol{q}}_i\right|^2 . \tag{2.1.20}$$

$\hat{\boldsymbol{p}}_i$ and $\hat{\boldsymbol{q}}_i$ are defined:

$$\begin{aligned} \hat{\boldsymbol{p}}_i &= \boldsymbol{p}_i - \boldsymbol{p}^*. \\ \hat{\boldsymbol{q}}_i &= \boldsymbol{q}_i - \boldsymbol{q}^*. \end{aligned} \tag{2.1.21}$$

Therefore, our core task is transformed into solving the matrix $\boldsymbol{M}$, and the computation of different transformation matrices $\boldsymbol{M}$ varies slightly. Later, for the function $l_v(\boldsymbol{x})$. let's define:

$$\boldsymbol{f}(\boldsymbol{v}) = l_v(\boldsymbol{v}). \tag{2.1.22}$$

2.1.2.3.1 Affine Transformation

$$E = \sum_i w_i \left(\hat{\boldsymbol{p}}_i \boldsymbol{M} - \hat{\boldsymbol{q}}_i\right)\left(\hat{\boldsymbol{p}}_i \boldsymbol{M} - \hat{\boldsymbol{q}}_i\right)^\top = \sum_i w_i \left(\hat{\boldsymbol{p}}_i \boldsymbol{M}\boldsymbol{M}^\top \hat{\boldsymbol{p}}_i{}^\top - \hat{\boldsymbol{q}}_i \boldsymbol{M}^\top \hat{\boldsymbol{p}}_i{}^\top - \hat{\boldsymbol{p}}_i \boldsymbol{M} \hat{\boldsymbol{q}}_i{}^\top + \hat{\boldsymbol{q}} \hat{\boldsymbol{q}}_i{}^\top\right). \tag{2.1.23}$$

Taking the derivative with respect to $\boldsymbol{M}$ on both sides, we have:

$$\frac{\partial E}{\partial \boldsymbol{M}} = 2\sum_i \omega_i (\hat{\boldsymbol{p}}_i{}^\top \hat{\boldsymbol{p}}_i \boldsymbol{M} - \hat{\boldsymbol{p}}_i{}^\top \hat{\boldsymbol{q}}_i) = 0. \tag{2.1.24}$$

Then

$$\boldsymbol{M} = (\sum_i \hat{\boldsymbol{p}}_i{}^\top \omega_i \hat{\boldsymbol{p}}_i)^{-1} \cdot (\sum_j \omega_j \hat{\boldsymbol{p}}_j{}^\top \hat{\boldsymbol{q}}_j). \tag{2.1.25}$$

$$\boldsymbol{f}_a(\boldsymbol{v}) = (\boldsymbol{v} - \boldsymbol{p}^*)(\sum_i \hat{\boldsymbol{p}}_i{}^\top \omega_i \hat{\boldsymbol{p}}_i)^{-1} \cdot (\sum_j \omega_j \hat{\boldsymbol{p}}_j{}^\top \hat{\boldsymbol{q}}_j) + \boldsymbol{q}^*. \tag{2.1.26}$$

Note that if interactivity is required, i.e., users can achieve real-time animation effects by changing the position of $\boldsymbol{q}$, for ease of computation, we can rearrange (2.1.26) as follows:

$$\boldsymbol{f}_a(\boldsymbol{v}) = \sum_j \boldsymbol{A}_j \hat{\boldsymbol{q}}_j + \boldsymbol{q}^*. \tag{2.1.27}$$

$\boldsymbol{A}_j$ is defined:

$$\boldsymbol{A}_j = (\boldsymbol{v} - \boldsymbol{p}^*)(\sum_i \hat{\boldsymbol{p}}_i{}^\top \omega_i \hat{\boldsymbol{p}}_i)^{-1} \cdot (\omega_j \hat{\boldsymbol{p}}_j{}^\top). \tag{2.1.28}$$

Note that the definition of $\boldsymbol{A}_j$ is independent of the value of $\boldsymbol{q}$, so it only needs to be computed once.

2.1.2.3.2 Similar Transformation However, affine transformations may result in uneven distribution or discontinuities, which are not typically observed in real-life objects. On the other hand, similarity transformations only involve translation, rotation, and uniform scaling. Therefore, we examine the mapping function under similarity transformations, where the core remains to find the matrix $\boldsymbol{A}$.

Similarity transformations require the matrix to possess the property $(\boldsymbol{M}^\top \boldsymbol{M} = \lambda^2 \boldsymbol{I})$. If we denote:

$$\boldsymbol{M} = (\boldsymbol{M}_1 \quad \boldsymbol{M}_2).$$

Then, $\boldsymbol{M}_1^\top \boldsymbol{M}_1 = \boldsymbol{M}_2^\top \boldsymbol{M}_2 = \lambda^2$ and $\boldsymbol{M}_1^\top \boldsymbol{M}_2 = 0$. They mean that

$$\boldsymbol{M}_2 = \boldsymbol{M}_1^\perp. \tag{2.1.29}$$

We can change the form of energy function:

$$E = \sum_i \omega_i \left| \begin{pmatrix} \hat{\boldsymbol{p}}_i \\ -\hat{\boldsymbol{p}}_i{}^\perp \end{pmatrix} \boldsymbol{M}_1 - \hat{\boldsymbol{q}}_i{}^\top \right|^2. \tag{2.1.30}$$

$$\frac{\partial E}{\partial \boldsymbol{M}_1} = 2 \sum_i \omega_i \begin{pmatrix} \hat{\boldsymbol{p}}_i \\ -\hat{\boldsymbol{p}}_i{}^\perp \end{pmatrix} \left(\begin{pmatrix} \hat{\boldsymbol{p}}_i{}^\top & -(\hat{\boldsymbol{p}}_i{}^\perp)^\top \end{pmatrix} \boldsymbol{M}_1 - \hat{\boldsymbol{q}}_i{}^\top \right) = 0. \tag{2.1.31}$$

We can get the value of $\boldsymbol{M}_1$:

$$\boldsymbol{M}_1 = \frac{1}{\mu_s} \sum_i \omega_i \begin{pmatrix} \hat{\boldsymbol{p}}_i \\ -\hat{\boldsymbol{p}}_i{}^\perp \end{pmatrix} \hat{\boldsymbol{q}}_i{}^\top. \tag{2.1.32}$$

$$\mu_s = \sum_i \omega_i \hat{\boldsymbol{p}}_i \hat{\boldsymbol{p}}_i{}^\top. \tag{2.1.33}$$

$$\boldsymbol{M} = \frac{1}{\mu_s} \sum_i \omega_i \begin{pmatrix} \hat{\boldsymbol{p}}_i \\ -\hat{\boldsymbol{p}}_i{}^\perp \end{pmatrix} \begin{pmatrix} \hat{\boldsymbol{q}}_i{}^\top & -(\hat{\boldsymbol{q}}_i{}^\perp)^\top \end{pmatrix} \tag{2.1.34}$$

Similarly, to enhance user interactivity, we can improve the expression as follows:

$$\boldsymbol{f}_s(v) = \sum_i \hat{\boldsymbol{q}}_i (\frac{1}{\mu_s} A_i) + \boldsymbol{q}^*. \tag{2.1.35}$$

$$\boldsymbol{A_i} = \omega_i \begin{pmatrix} \hat{\boldsymbol{p}}_i \\ -\hat{\boldsymbol{p}}_i{}^\perp \end{pmatrix} \begin{pmatrix} \boldsymbol{v} - \boldsymbol{p}^* \\ -(\boldsymbol{v} - \boldsymbol{p}^*)^\perp \end{pmatrix}^\top. \tag{2.1.36}$$

We can find that A_i is only up to $\boldsymbol{p}$ and $\boldsymbol{v}$.

Similarity transformations have a good property of preserving angles, thus resulting in more ideal outcomes compared to affine transformations. However, they may lead to the enlargement of the range for parts of points that are distant from the transformation points.

2.1.2.3.3 Rigid Transformation In previous work, it was found that achieving more realistic deformations requires approaching rigid transformations as closely as possible. The author proposed Lemma 2.1 and provided the following concise expression:

$$\boldsymbol{f}_r(\boldsymbol{v}) = \sum_i \hat{\boldsymbol{q}}_i \boldsymbol{A}_i. \tag{2.1.37}$$

$$\boldsymbol{f}_r(\boldsymbol{v}) = |\boldsymbol{v} - \boldsymbol{p}^*| \frac{f_r(\boldsymbol{v})}{|f_r(\boldsymbol{v})|} + \boldsymbol{q}^*. \tag{2.1.38}$$

$\boldsymbol{f}_r(\boldsymbol{v})$ is the sought-after rigid transformation function. Compared to similarity transformations, rigid transformations utilize normalization, thus leading to increased computational complexity.

2.2 Poisson Editing

2.2.1 Problem Statement

Image editing tasks concern either global changes (color/intensity corrections, filters, deformations) or local changes confined to a selection. Here we are interested in achieving local changes, ones that are restricted to a region manually selected, in a seamless and effortless manner. The extent of the changes ranges from slight distortions to complete replacement by novel content. Classic tools to achieve that include image filters confined to a selection, for slight changes, and interactive cut-and-paste with cloning tools for complete replacements. With these classic tools, changes in the selected regions result in visible seams, which can be only partly hidden, subsequently, by feathering along the border of the selected region.

We propose here a generic machinery from which different tools for seamless editing and cloning of a selection region can be derived. The mathematical tool at the heart of the approach is the Poisson partial differential equation with Dirichlet boundary conditions which specifies the Laplacian of an unknown function over the domain of interest, along with the unknown function values over the boundary of the domain. The motivation is twofold.

The essence of this problem is essentially an interpolation task: inserting pixels from the source image into the selected region of the target image to achieve a seamless effect. Our requirement is to minimize gradient changes, which transforms the problem into a Poisson equation problem with given boundaries. After discretization, it can be converted into a sparse matrix solving problem. Once the pixel values are obtained, they are drawn onto the target image.

2.2.2 Propose Algorithms

Let S be a closed subset of $\mathbb{R}^2$. It represents the image definition domain. Let Ω be a be a closed subset of S with boundary $\partial\Omega$. Let f^* be a known scalar function defined over S minus the interior of Ω and let f be an unknown scalar function defined over the interior of Ω. Finally, let $\boldsymbol{v}$ be a vector field defined over Ω.

The simplest interpolant f of f^* over Ω is the membrane interpolant defined as the solution of the minimization problem:

$$\min_f \iint_\Omega |\nabla f|^2 \text{ with } f|_{\partial\Omega} = f^*|_{\partial\Omega}. \tag{2.2.1}$$

The minimizer must satisfy the associated Euler-Lagrange equation.

$$\Delta f = 0 \text{ over } \Omega \text{ with } f|_{\partial\Omega} = f^*|_{\partial\Omega}. \tag{2.2.2}$$

According to this principle, we give the discretization of equations. For all $p \in \Omega$,

$$|N_p| f_p - \sum_{q \in N_p \cap \Omega} f_q = \sum_{q \in N_p \cap \partial\Omega} f_q^* + \sum_{q \in N_p} v_{pq}. \tag{2.2.3}$$

In equation 2.2.3, N_p represents a four-connected neighbor. We realize too different methods, they are only different from the value of v_{pq}. The definitions of v_{pq} are listed:

1. Seamless:

$$v_{pq} = g_p - g_q;$$

2. mixed:

$$v_{pq} = \begin{cases} f_p^* - f_q^* & \text{if } |f_p^* - f_q^*| > |g_p - g_q|, \\ g_p - g_q & \text{otherwise,} \end{cases}$$

In this way, we can construct the sparse matrix equation according to 2.2.3.

2.3 Tutte Parameterization

2.3.1 Problem Statement

First, read the Framework3D framework configuration instructions and configure the framework. Currently, there is no need to delve into the implementation of the framework. Simply refer to the documentation and examples to understand how to write a single node .cpp file.

Second, Read about mesh data structures to learn the basic representation of meshes, as well as the OpenMesh library we will use. An example using the homework framework and half-edge structure to compute and visualize curvature of triangular meshes is provided. Besides, Learn how to traverse vertices, faces, edges, and access their neighborhoods using the half-edge structure. The structure is shown in Figure 2.1.

Then, refer to papers and assignment requirements to implement the Tutte surface parameterization algorithm. We can add node implementations for the algorithm in the specified directory and test the

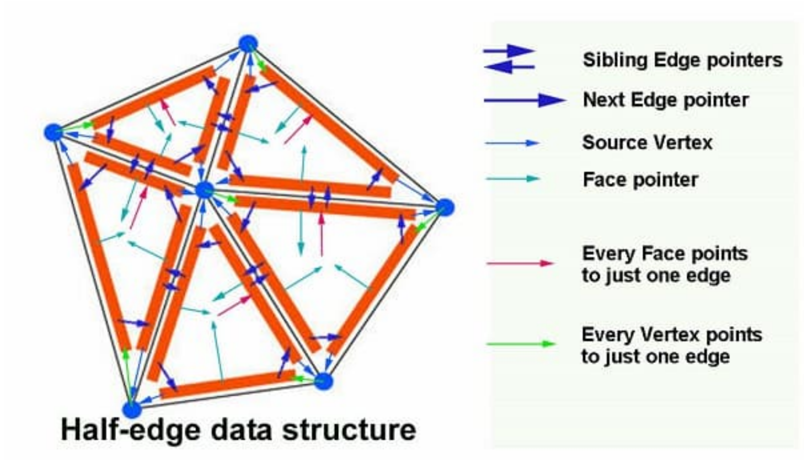


Figure 2.1: half-edge structure

correctness of the algorithm in the framework, as well as the impact of different variable selections on experimental results. We can complete this assignment following the steps below:

1. Implement the establishment and solution of the Laplace equation on the mesh (with uniform weights). The boundary conditions are still chosen as the original spatial point positions, and you can obtain a "minimal surface" with fixed boundaries by solving it, completing the minimal surface node;
2. Under the same coefficients, modify the boundary conditions to map the mesh boundary to the boundary of a convex area on the plane (square, circle), completing the boundary mapping node;
3. Solve for (uniformly weighted) Tutte parameterization (connecting the above two nodes!), then use the texture mapping node to visualize parameter coordinates;
4. Try and compare different weight selections, such as Cotangent weights (to be implemented) and shape-preserving weights (Floater weights).

2.3.2 Propose Algorithms

First, we need to define vertex differentials:

$$\boldsymbol{\delta}_i = \boldsymbol{v_i} - \sum_{j \in N(i)} w_j \boldsymbol{v_j}, \tag{2.3.1}$$

In equation 2.3.1, $N(i)$ represents the 1-neighborhood of i. w_i represents the weight of i. We have two definitions of w_i:

1. Uniform weights:: $w_j = 1$
2. Cotangent weights: $w_j = \cot \alpha_{ij} + \cot \beta_{ij}$, the definitions of α_{ij} and β_{ij} are shown in Figure 2.2;

However, we need to notice that the coefficients should be normalized:

$$w_j = \frac{w_j}{\sum\limits_k w_k}. \tag{2.3.2}$$

Then, we can calculate the minimal surface by constructing linear equations. Let $\boldsymbol{\delta}_i = \mathbf{0}$, we know

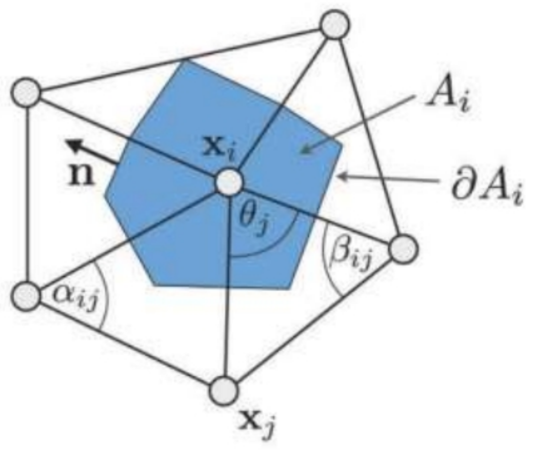


Figure 2.2: α_{ij} and β_{ij}

that

$$\boldsymbol{v_i} - \sum_{j \in N(i)} w_j \boldsymbol{v_j} = \mathbf{0}. \tag{2.3.3}$$

The equation 2.3.3 gives the restriction to all interior points, to simplify the construction of linear equation, the objective of establishing the equation can be modified to include all types of points. For the boundary points, the restriction is

$$\boldsymbol{v_i} = \boldsymbol{v_i}. \tag{2.3.4}$$

The right side of equation is given. This way of construction linear equations is much easier. It also does not affect computational efficiency, as boundary points are always sparse compared to the overall set of points.

Then, we can also realize the boundary mapping. First, we need to find the boundary. I write $halfedge\_handle\_start$ to find handle on the boundary, and use this handle to find all the boundary points.

Two different kinds of boundaries are provided.

1. Circle in [0,1]×[0,1];
2. Square [0,1]×[0,1].

The most important problem to be solved is to determine distribution of boundary points. A simple approach is to evenly distribute boundary points along the circumference, but it's obviously not realistic. My approach is to define initial points and then calculate the arc length distance from each point to the starting point in sequence, storing it in $bp\_distance$, and then distribute them according to the proportion of this distance to the total arc length. To be more specific, the relationship between coordinates and distances is as follows:

1. Circle: the starting point is $(1,\frac{1}{2})$.

$$x = \frac{1}{2} + \frac{1}{2}\cos(\frac{dis}{sum} * 2\pi), \quad y = \frac{1}{2} + \frac{1}{2}\sin(\frac{dis}{sum} * 2\pi) \tag{2.3.5}$$

2. Square: the starting point is (0,0).

$$x = \begin{cases} 4 * \frac{dis}{sum}, & \frac{dis}{sum} \in [0, \frac{1}{4}) \\ 1, & \frac{dis}{sum} \in [\frac{1}{4}, \frac{1}{2}) \\ 3 - 4 * \frac{dis}{sum}, & \frac{dis}{sum} \in [\frac{1}{2}, \frac{3}{4}) \\ 0. & \frac{dis}{sum} \in [\frac{3}{4}, 1) \end{cases} \qquad y = \begin{cases} 0, & \frac{dis}{sum} \in [0, \frac{1}{4}) \\ 4 * \frac{dis}{sum} - 1, & \frac{dis}{sum} \in [\frac{1}{4}, \frac{1}{2}) \\ 1, & \frac{dis}{sum} \in [\frac{1}{2}, \frac{3}{4}) \\ 4 - 4 * \frac{dis}{sum}. & \frac{dis}{sum} \in [\frac{3}{4}, 1) \end{cases} \tag{2.3.6}$$

We can assume the boundary points are all in x-y plain, so the z-coordinates are zero. We just use the new boundary coordinates to replace the right side of equation 2.3.4, then get the construction of boundary mapping.

Finally, we use SparseLU to solve the equation and *set_point* function to reconstruct the 3D object.

2.3.3 Programming

2.3.3.1 Halfedge Structure

It is shown in Figure 2.1. Then OpenMesh has provided us many functions to read the information from the mesh structure.

```
for (const auto& vertex_handle : halfedge_mesh->vertices())
```

This loop can obtain vertex-related information.

```
for (const auto& halfedge_handle : vertex_handle.outgoing_halfedges())
```

This loop can obtain half-edge information related to a vertex. Some functions used are shown as follows:

1. halfedge_handle.to(): use the halfedge to find the end point;
2. halfedge_handle.prev(): find the last handle;
3. halfedge_handle.opp(): find the opposite halfedge handle;
4. is_boundary(): judge whether the halfedge and vertex is on the boundary;
5. .idx(): find the index of vertex.

2.3.3.2 Node programming

Another interesting task is completing node programming. Node programming refers to the process of writing code or scripts that operate on individual nodes within a system or network. In the context of software development or computer science, nodes typically represent individual elements or entities within a larger system, such as data structures, computational units, or network devices.

Benefits of Node Programming:

1. Modularity: Node programming allows for the development of modular and scalable systems, where individual nodes can be easily added, removed, or modified without affecting the entire system.

2. Concurrent Processing: Nodes can often operate concurrently, allowing for parallel execution of tasks and improved performance in multi-core or distributed systems.
3. Flexibility: Node programming provides flexibility in system design and implementation, as developers can choose the appropriate level of granularity for nodes based on the requirements of the application.
4. Fault Isolation: By encapsulating functionality within individual nodes, node programming can help isolate faults and minimize the impact of failures on the overall system.

Drawbacks of Node Programming:

1. Complexity: Managing a large number of nodes and their interactions can introduce complexity into the system, leading to potential challenges in debugging, testing, and maintenance.
2. Coordination Overhead: In distributed systems, coordinating the behavior of multiple nodes may introduce overhead and latency, especially when nodes need to communicate and synchronize with each other.
3. Resource Consumption: Each node consumes system resources such as memory, processing power, and network bandwidth. Managing resource usage across a large number of nodes can be challenging and may impact overall system performance.
4. Scalability Challenges: While node programming can support scalability, achieving efficient scaling in practice may require careful design and optimization to avoid bottlenecks and resource contention.

2.4 ARAP Parameterization

2.4.1 Problem Statement

In this experiment, we aim to achieve parameterization with no fixed boundaries. In the previous experiment, we have already implemented the mapping of a grid to a given boundary, where the generation of the entire grid involves solving a sparse system of equations. Now, we need to consider how to minimize the deformation of three-dimensional triangles, i.e., minimizing an energy function that measures distortion. The work being replicated in this paper is Ligang Liu's work from 2008. Additionally, we need to continue learning about the usage of the Eigen library, establishing and solving sparse matrices, SVD decomposition, and node programming.

2.4.2 Propose Algorithms

Firstly, we need to consider the general principles of the algorithm for non-fixed boundary parameterization, which involves minimizing the following energy function:

$$E(u, L) = \frac{1}{2}\sum_{t=1}^{T}\sum_{i=0}^{2}\cot(\theta_t^i)||u_t^i - u_t^{i+1} - L_t(x_t^i - x_t^{i+1})||^2. \tag{2.4.1}$$

In this expression, T represents the set of all triangles in the mesh, i iterates over each vertex, $\cot(\theta_t^i)$ measures the weight of the area, u_t^i represents the 2D coordinates of the i-th vertex of the t-th triangle (chosen as a plane in the initial parameterization), L_t represents the distortion of the t-th triangle (computed by transforming the 3D triangle into an equivalent 2D triangle and calculating the Jacobian matrix), and x_t^i represents the 2D coordinates obtained after mapping the 3D triangle in a distortion-minimizing manner.

Therefore, the essence of non-fixed boundary parameterization is to find suitable (u, L) that satisfy equation 2.4.2 as much as possible.

$$(u, L) = argmin_{(u,L)} E(u, L). \quad L_t \in M. \tag{2.4.2}$$

Where M is the desired linear transformation.

In this paper, we have reproduced the ARAP (As-Rigid-As-Possible) algorithm, which requires the matrix M to satisfy the following requirements:

$$M = \left\{ \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} : \theta \in [0, 2\pi) \right\}$$

In fact, this is a non-linear optimization problem, so it cannot be directly solved by taking partial derivatives. Therefore, it requires the introduction of local/global methods. The general steps are as follows.

1. Implement an initial parameterization: This step can utilize any parameterization result obtained from your work in Assignment 4.
2. Implement the local iteration step of ARAP (Local Phase): Independently execute on each triangle t, relatively straightforward. Implement a second-order matrix SVD decomposition. Fix parameter coordinates, compute the local orthogonal approximation of the current parameterization Jacobian.
3. Implement the global iteration step of ARAP (Global Phase): Fix on each triangle t, update parameter coordinates, solve a global sparse linear system of equations. The coefficients of the equation system remain fixed and only need to be set once and pre-decomposed.
4. Iterate several times and observe the results.

To implement Step 1, I added a new interface in the interactive interface, allowing the input of the initialized geometry, which can be connected to the boundary mapping results obtained in Assignment 4. It is shown in Figure 2.3.

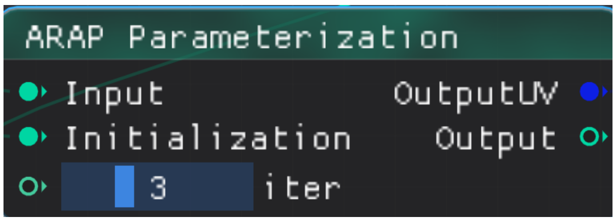


Figure 2.3: Node ARAP

”Input” refers to the initial geometry passed in, while ”Initialization” refers to the initialized geometry, which can include the boundary mapping result from the fourth assignment. ”iter” can be used to set the number of iterations. ”Output” refers to the geometry after flattening.

Afterwards, it is necessary to emphasize the process of flattening, and we assume that the result of the flattening is as follows:

$$x_1(0,0,0),\ x_2(|v_1v_2|,0,0),\ x_3(|v_1v_3|\cos\theta,|v_1v_3|\sin\theta,0).$$

Here, θ represents the angle at vertex 1, and the part related to v represents the triangular mesh faces of the 3D triangle.

For the ”Local” stage, keeping the initialization fixed (i.e., halfedge_mesh_ini), for each triangular mesh t, we solve for the L_t that satisfies the conditions. Here, we use the following SVD decomposition:

$$S_t(u) = \sum_{i=0}^{2} \cot\theta_t^i (u_t^i - u_t^{i+1})(x_t^i - x_t^{i+1})^T = USV^T. \tag{2.4.3}$$

$$L_t = UV^T. \tag{2.4.4}$$

L_t represents the optimal approximation rotation matrix that implies deformation.

For the ”Global” stage, we fix L_t and solve for the optimized mesh coordinates $\boldsymbol{u}$, which essentially involves solving a sparse matrix.

$$\sum_{j\in N(i)} (\cot\theta_{ij}+\cot\theta_{ji})(u_i-u_j) = \sum_{j\in N(i)} (\cot\theta_{ij}L_{t(i,j)}+\cot\theta_{ji}L_{t(j,i)})(x_i-x_j)\ \forall i=1,\cdots,n. \tag{2.4.5}$$

Where θ_{ij} represents the dihedral angle between half-edge ij in the mesh, which can be obtained through some operations on the half-edge data structure.

This way, we can obtain the new optimized mesh. Then, we utilize the custom function ”Update_” to perform updates, and proceed to the ”Local” stage for iteration until reaching the specified number of steps.

2.5 Phong Shading

2.5.1 Problem Statement

We want to implement local rendering with the Blinn-Phong shading model, and to enhance the realism of the scene, we also want to add shadows using the Shadow Mapping algorithm. By implementing these two algorithms, we aim to achieve a preliminary rendering model.

2.5.2 Propose Algorithms

2.5.2.1 Blinn-Phong Shading Model

The core expression of the Blinn-Phong shading model, as shown in Equation 2.5.1, is what we aim to implement first.

$$I = k_a I_a + k_d I_d \cos\alpha + k_s I_s \cos^k\theta. \tag{2.5.1}$$

Where θ represents the angle between the reflected light and the viewing direction, and α represents the angle between the incident light and the surface normal. The remaining coefficients are defined as follows:

1. k_a represents the ambient coefficient, with $k_a = 0.2$ in this case.
2. k_s represents the specular reflection coefficient, with $k_s = metallic * 0.8$.
3. k_d represents the diffuse reflection coefficient, with $k_d = 1 - k_s$.
4. $k = (1 - roughness) * 0.2$.

For multiple light sources, we can simply accumulate the contributions from each light source.

2.5.2.2 Shadow Mapping

If there is occlusion between the shading point and the light source, naturally the shading point should appear as ”shadowed”, and the intensity of light at this point should only come from ambient light. This can significantly reduce computational complexity and make the scene more realistic.

The core idea of this algorithm is that if the actual distance from the shading point to the light source is greater than the depth obtained by capturing the scene from the light source’s perspective, it indicates that the shading point is occluded and thus becomes shadowed. We only need to use one condition to implement this problem.

2.5.2.3 Normal Mapping

In our scene, polygonal objects are abundant, each possibly composed of hundreds or thousands of flat triangles. To enhance realism and conceal the fact that these objects are made up of numerous triangles, we apply textures to the triangles to add extra details. While textures are beneficial, the fact that objects are composed of many triangles becomes apparent upon close inspection. In reality, object surfaces are not perfectly flat but exhibit countless details and irregularities. Normal maps preserve the characteristics of each fragment, allowing for significant enhancements in areas with details.

Since the normal map stores normals in tangent space, it needs to be transformed into world coordinates using a transformation matrix. This transformation matrix is called the tangent-to-world matrix and is composed of the tangent, bitangent, and normal vectors at the point of interest. These three vectors are mutually orthogonal unit vectors. The specific process is as follows:

```
normal = mat3 (tangent,bitangent,normal)*(2*normalmap_value-1);
```

2.6 Path Tracing

2.6.1 Problem Statement

In our last task, we implemented a model for local lighting and the rendering of shadows. However, such a lighting model is evidently very limited. The lighting at a single point is highly complex, and the influence of light from different types and directions on colors is also extremely intricate. Therefore, in this task, we need to implement a more sophisticated rendering model, which will require numerous complex processing techniques.

2.6.2 Propose Algorithms

2.6.2.1 Physics Based Rendering

To produce images with lighting and shadows that closely resemble the real world, we need to consider how light interacts with objects in the physical world and then enters the human eye, creating the colorful and beautiful world we perceive. Due to space constraints, we will skip the introduction of geometric optics and wave optics knowledge, and only introduce some physical concepts in light rendering techniques.

2.6.2.1.1 Irradiance The irradiance $L(x, \omega)$ of a light ray is defined as the power per unit area, where x is the spatial position and ω is the direction along which it is measured. The irradiance can be denoted symbolically.

2.6.2.1.2 BRDF Function In the real world, objects of different materials exhibit a wide variety of reflection patterns when interacting with light. To describe how materials reflect incident light at a given spatial position, we use the Bidirectional Reflectance Distribution Function (BRDF) $f_r(x, \omega_i, \omega_o)$.

2.6.2.1.3 Rendering Equation The rendering equation describes the radiance at a point on a surface in a scene. It states that the outgoing radiance at a point in a given viewing direction is equal to the sum of emitted radiance at that point and the incoming radiance from all directions adjusted by the BRDF function. This equation is crucial for rendering images as it describes how light propagates and interacts within a scene.

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_r(x, \omega_i, \omega_o) \cdot L_i(x, \omega_i) \cdot (\omega_i \cdot n)\, d\omega_i \tag{2.6.1}$$

- $L_o(x, \omega_o)$ is the outgoing radiance from point x in the direction ω_o,

- $L_e(x, \omega_o)$ is the emitted radiance at point x in the direction ω_o,
- $f_r(x, \omega_i, \omega_o)$ is the Bidirectional Reflectance Distribution Function (BRDF) at point x for incoming direction ω_i and outgoing direction ω_o,
- $L_i(x, \omega_i)$ is the incoming radiance at point x from direction ω_i,
- $(\omega_i \cdot n)$ is the cosine factor, where n is the surface normal,
- Ω represents the hemisphere of incoming directions.

Interacting light rays with the scene is crucial, as illustrated in Figure 2.4.

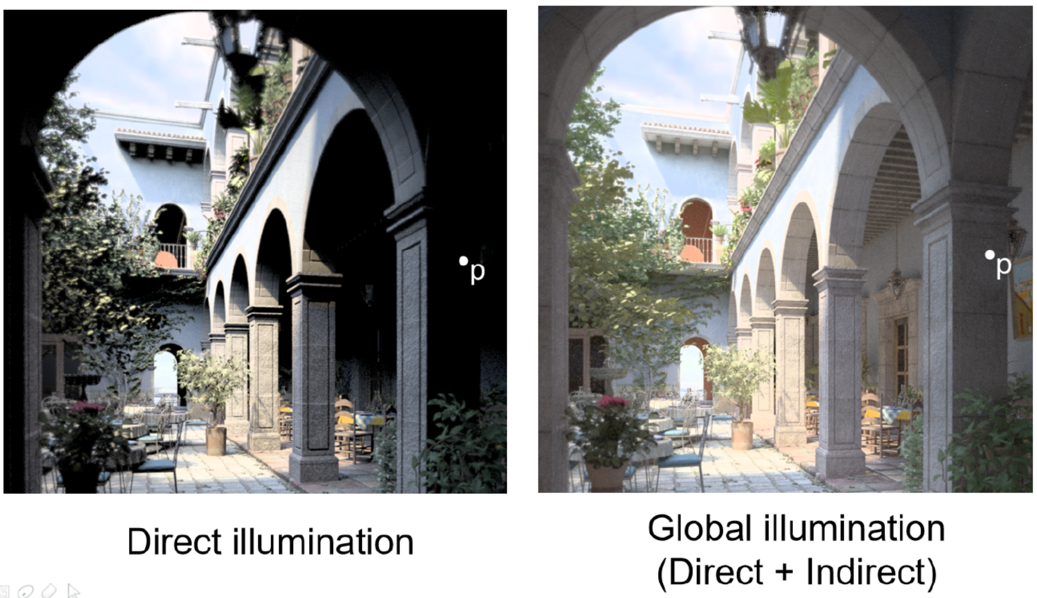


Figure 2.4: Interacting Light Rays with Scene

In the case of global illumination alone, as depicted, the scene appears dim, with many details of objects not directly illuminated by the light source hidden in darkness. However, with the combination of global illumination and local illumination, the overall scene becomes brighter, and it can illuminate details of objects that were not visible in the left image.

2.6.2.2 Ray Tracing

The ray tracing algorithm is an important early rendering method in computer graphics. As the name suggests, its core idea is to trace rays of light from the camera to find light sources. Specifically, a ray of light is cast from each pixel on the camera plane. This ray undergoes a series of reflections/refractions in the scene before eventually reaching either a light source or the background. Light that reaches the background at the pixel presents the color of the corresponding position on the background (multiplied by a series of factors along the path), while light reaching the light source presents the color of the light source at that point (also multiplied by a series of factors along the path). The core principle is illustrated in Figure 2.5.

The main steps of the original algorithm (Whitted-style) are as follows:

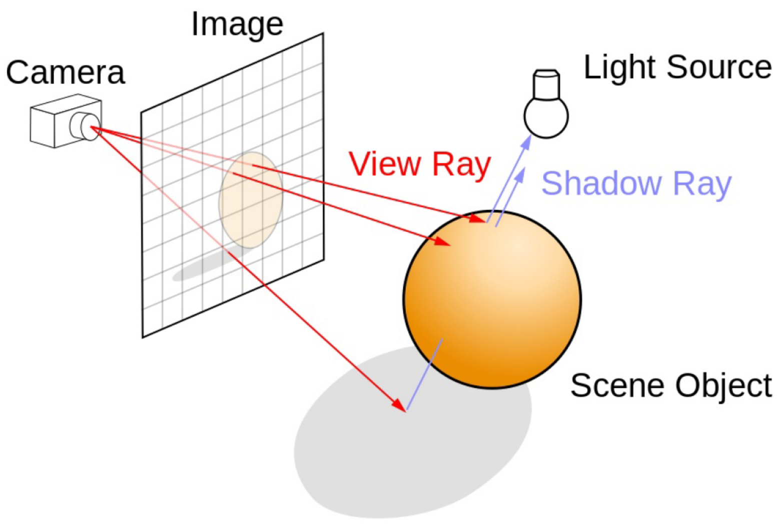


Figure 2.5: Ray Tracing

```
color trace(point p, vector d, int step)
{
    color local, reflected, transmitted;
    point q;
    normal n;
    if(step > max) return(background_color);
    q = intersect(p,d,status);
    if(status == light_source)
    return(light_source_color);
    if(status == no_intersection)
    return(background_color);
    n = normal(q);
    r = reflect(q,n);
    t = transmit(q,n);
    local = phong(q,n,r);
    reflected = trace(q,r,step+1);
    transmitted = trace(q,t,step+1);
    return(local+reflected+transmitted);
}
```

The main shortcomings of the ray tracing algorithm are as follows:

1. When the light source area is small, the probability of rays emitted from the camera reaching the light source decreases, leading to many rays being wasted as they cannot reach the light source, resulting in a large number of noise points. As shown in the Figure 2.6.
2. The ray tracing algorithm actually does not solve the rendering equation; instead, it is based on simple principles of ray reflection/refraction. Therefore, the physical accuracy of the results cannot be guaranteed.

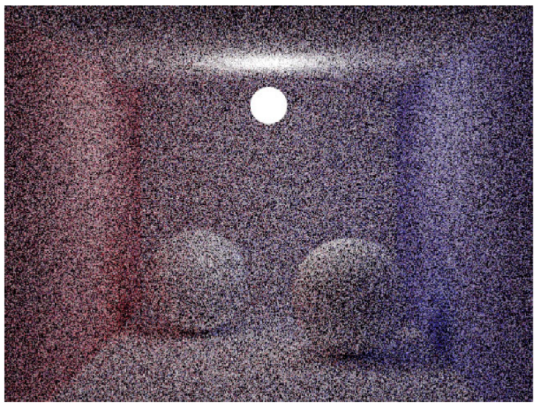


Figure 2.6: Ray Tracing Problem

2.6.2.3 Monte Carlo sampling method

The Monte Carlo sampling method is a numerical computation method based on probability statistics, commonly used to simulate random phenomena. In graphics, Monte Carlo methods are often used to estimate complex lighting effects in ray tracing, such as global illumination and indirect lighting.

This approach approximates solutions by generating a large number of random samples, thereby compensating for the shortcomings of traditional ray tracing algorithms to some extent. Monte Carlo sampling can effectively reduce noise and improve the quality and realism of rendering results.

The description of the problem is as follows:

We need to solve the numerical integral as follows:

$$I = \int_a^b f(x)\,dx.$$

We can estimate the value of this integral using a computer through sampling methods. Our goal is to construct random variables to obtain an unbiased estimate, denoted as

$$\hat{I} = \frac{f(x)}{p(x)}.$$

In fact,

$$E[\hat{I}] = E[\frac{f(x)}{p(x)}] = \int_a^b \frac{f(x)}{p(x)} p(x)\,dx = \int_a^b f(x)\,dx$$

This way, we obtain an unbiased estimate of $\hat{I}$.

However, in practice, we often do not know the probability distribution function, only can estimate it. The most commonly used estimations include the uniform distribution. Suppose we sample n samples, $x_1, x_2, \ldots, x_n$,

$$\hat{I}_j = \frac{f(x_j)}{p(x_j)}.$$

Then we obtain an estimate of the original integral as follows:

$$\bar{I} = \frac{1}{n}\sum_{j=1}^{n}\hat{I}_j.$$

This estimate satisfies:

$$\int_a^b f(x)\,dx = \lim_{n\to\infty}\bar{I} = \lim_{n\to\infty}\frac{1}{n}\sum_{j=1}^{n}\hat{I}_j.$$

We can also introduce parameters p_i according to the importance of different sampling points, satisfying:

$$\sum_{j=1}^{n}\omega_j = 1.$$

Finally:

$$\bar{I} = \sum_{j=1}^{n}\omega_j\frac{f(x_j)}{p(x_j)}.$$

This is the core idea of using Monte Carlo sampling algorithm to solve numerical integration.

2.6.2.4 Pace Tracing

Path tracing algorithm is a ray tracing method based on the Monte Carlo sampling algorithm. Its core idea is similar to ray tracing, which is to trace rays from the camera to find light sources. However, in path tracing, multiple rays are emitted from each pixel on the camera plane to search for light sources. When rays intersect with objects in the scene and undergo reflection, a direction is sampled from the hemisphere according to a predetermined probability distribution function (referred to as BRDF sampling), and this probability factor is then removed from the final expression. The rest of the process is consistent with ray tracing.

The essence of the path tracing algorithm is to combine the contributions of multiple light paths to the lighting effect according to their probabilities, thus realizing the rendering equation. The core algorithm is shown in Figure 2.7.

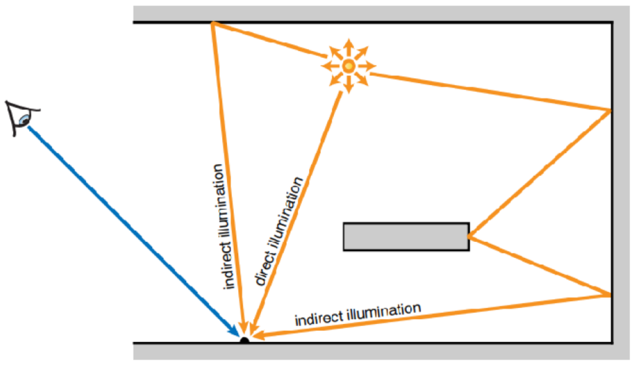


Figure 2.7: Path Tracing

It's worth mentioning that since this is a recursive algorithm, it must terminate. To ensure termination of the iterative reflection of rays, we can introduce the technique of Russian Roulette: generating a random number at each iteration to determine whether to proceed with the next reflection of the ray, thus preventing the possibility of infinite iteration.

The algorithm proceeds as follows:

```
shade(p, wo)
    Test Russian Roulette with p_rr, if fail then return 0.0;
    Randomly choose one direction wi with pdf(wi);
    Trace a ray r(p, wi);
    if ray r hit the light:
        return L_i * f_r * cos / pdf(wi) / p_rr;
    else if ray r hit an obj at position q:
        return shade(q, -wi) * f_r * cos / pdf(wi) / p_rr;
ray_generation(camPos, pixel)
    Uniformly choose N sample postitions within pixel;
    pixel_radience = 0.0;
    for each sample in the pixel:
        Shoot a ray r(camPos, cam_to_sample_direction);
        If ray r hit the scene at p:
        pixel_radience += 1/N * shade(p, cam_to_sample_direction);
    return pixel_radience;
```

However, it is worth noting that the path tracing algorithm does not solve the problem when the light source area is very small, and emitted rays cannot find the light source. A natural approach to address this issue is: since rays emitted from the camera cannot reach the light source, can we directly emit rays from the light source to reach objects and directly calculate global illumination? Following this idea, we can improve the efficiency of the path tracing algorithm by sampling the light source. As shown in the following Figure 2.8.

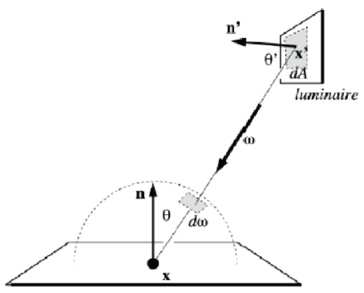


Figure 2.8: Path Tracing Problem and Solution

At this point, by geometric relationships, the integral over the BRDF hemisphere is transformed to the plane A where the area light source is located:

$$L_0(x,\omega_0) = \int_\Omega L_i(x,\omega_i) f_r(x,\omega_i,\omega_o)\cos\theta \mathrm{d}\omega_i = \int_A L_i(x,\omega_i) f_r(x,\omega_i,\omega_o)\frac{\cos\theta\cos\theta'}{||x'-x||^2}\mathrm{d}A. \quad (2.6.2)$$

In the improved path tracing algorithm, the main process of shading is as follows:

```
shade(p, wo)
   // direct light, sampling the light
   Uniformly sample the light at x with pdf_light = 1/A;
   Check visibility, if invisible L_dir = 0;
   else L_dir = L_i * f_r * cos(theta1) *
cos(theta2) / |x - p|^2 / pdf_light;

   // indirect light, BRDF sampling
   L_indir = 0.0;
   Test Russian Roulette with p_rr
   Randomly choose one direction wi with pdf(wi);
   Trace a ray r(p, wi);
   if ray r hit an non-emitting obj at position q:
      return shade(q, -wi) * f_r * cos / pdf(wi) / p_rr;
   return L_dir + L_indir;
```

2.7 Mass Spring

2.7.1 Problem Statement

This assignment involves simulation, specifically using a spring-mass system. The spring-mass system is a fundamental technique in elastic body simulation. Due to its ease of implementation and good results, it has been widely used in applications such as gaming for simulating hair, cloth, and elastic bodies. It has also inspired many subsequent methods for elastic body simulation.

A spring-mass system is essentially a graph composed of nodes and edges between them. Each node of the graph represents a mass, and each edge represents a spring. It discretizes the continuous body. The grid can be a 2D grid, used for simulating objects like cloth or paper, as shown in the figure below. It can also be a 3D grid used for simulating volumetric objects.

2.7.2 Propose Algorithms

2.7.2.1 Semi-Implicit Method

To get the discrete motion of these points over time, we need to discretize time. Suppose we have n vertices, and we arrange all vertices into a matrix $\mathbf{x} \in \mathbb{R}^{3n\times 1}$.

To initiate the movement of the object, one approach is to assign a velocity to each vertex. If we use semi-implicit time integration, the velocity $\mathbf{v}^{n+1} \in \mathbb{R}^{3n\times 1}$ can be determined as follows, according to Newton's second law:

$$\mathbf{v}^{n+1} = \mathbf{v}^n + h\mathbf{M}^{-1}(\mathbf{f}_{\text{int}}(\mathbf{x}^n) + \mathbf{f}_{\text{ext}})$$

Here, $\mathbf{M} \in \mathbb{R}^{3n\times 3n}$ is the system's mass matrix. Here, we simply set it as a diagonal matrix.

According to the energy perspective, let E be the elastic energy of the system. The internal elastic force $\mathbf{f}_{\text{int}}$ can be defined as:

$$\mathbf{f}_{\text{int}} = -\nabla E$$

If we have an expression for the energy gradient, we can calculate the internal elastic force. By adding external forces such as gravity, we can obtain the total force acting on the vertices. Then, we can initiate the movement of the object!

Now, let's define the energy of a spring i as:

$$E_i = \frac{k}{2}(\|\mathbf{x}_{i1} - \mathbf{x}_{i2}\| - L)^2$$

We define $\mathbf{x}_i := \mathbf{x}_{i1} - \mathbf{x}_{i2}$. Then, the total energy is:

$$E = \sum_i E_i = \sum_i \frac{k}{2}\left(\|\mathbf{x}_i\| - L\right)^2$$

Its gradient can be calculated as:

$$\nabla E = \sum_i k(\|\mathbf{x}_i\| - L)\frac{\mathbf{x}_i}{\|\mathbf{x}_i\|}$$

Finally, to simulate damping, we can multiply the velocity by a damping coefficient vel $\times$ damping.

Here, we need to fix some points (Dirichlet boundary conditions). Then, we can simply set the external forces and velocities of these fixed points to zero.

2.7.2.2 Implicit Euler Integration

Next, we need to implement implicit Euler integration:

$$\begin{aligned}\mathbf{x}^{n+1} &= \mathbf{x}^n + h\mathbf{v}^{n+1}\\ \mathbf{v}^{n+1} &= \mathbf{v}^n + h\mathbf{M}^{-1}(\mathbf{f}_{\text{int}}(\mathbf{x}^{n+1}) + \mathbf{f}_{\text{ext}})\end{aligned}$$

But we find that this problem needs to be turned into an equation about $\mathbf{x}^{n+1}$.

Here is a commonly used approach to solve it.

Rearranging:

$$\mathbf{x}^{n+1} = \mathbf{x}^n + h\mathbf{v}^n + h^2\mathbf{M}^{-1}(-\nabla E(\mathbf{x}^{n+1}) + \mathbf{f}_{\text{ext}}) \tag{4}$$

Let's define $\mathbf{y} := \mathbf{x}^n + h\mathbf{v}^n + h^2\mathbf{M}^{-1}\mathbf{f}_{\text{ext}}$.

We can transform equation (4) into:

$$\frac{1}{h^2}\mathbf{M}(\mathbf{x}^{n+1} - \mathbf{y}) + \nabla E(\mathbf{x}^{n+1}) = \mathbf{0}$$

This equation can be viewed as the first-order optimality condition (KKT) for an optimization problem (let $\mathbf{x} = \mathbf{x}^{n+1} \in \mathbf{R}^{3n\times 1}$):

$$\min_{\mathbf{x}} \quad g(\mathbf{x}) = \frac{1}{2h^2}(\mathbf{x} - \mathbf{y})^\top \mathbf{M}(\mathbf{x} - \mathbf{y}) + E(\mathbf{x}) \tag{5}$$

This brings us back to the optimization field that you may have (probably) studied.

The derivative of energy g is:

$$\nabla g(\mathbf{x}) = \frac{1}{h^2}\mathbf{M}(\mathbf{x} - \mathbf{y}) + \nabla E(\mathbf{x})$$

To solve the optimization problem, we can use gradient descent, but its convergence speed is relatively slow (linear convergence rate). In computer graphics, a more commonly used approach is to use Newton's method:

$$\mathbf{x}^{n+1} = \mathbf{x}^n - \mathbf{H}^{-1}\nabla\mathbf{g}$$

So we need to compute the Hessian matrix $\mathbf{H} = \nabla^2 g$ of energy g.

First, let's look at the Hessian of a spring's energy:

$$\begin{aligned}\mathbf{H}_i &= \nabla^2 E_i \\ &= k\frac{\mathbf{x}_i\mathbf{x}_i{}^\top}{\|\mathbf{x}_i\|^2} + k\left(1 - \frac{L}{\|\mathbf{x}_i\|}\right)\left(\mathbf{I} - \frac{\mathbf{x}_i\mathbf{x}_i^{\mathrm{T}}}{\|\mathbf{x}_i\|^2}\right)\end{aligned}$$

So the overall Hessian $\mathbf{H} \in \mathbf{R}^{3n\times 3n}$ is assembled by stacking the Hessian of each spring $\mathbf{H} \in \mathbf{R}^{3\times 3}$ according to vertex indices. The def of H is shown in Figure 2.9.

Finally, we write out the Hessian of g:

$$\mathbf{H}(\mathbf{x}) = \sum_{e=\{i,j\}} \begin{bmatrix} \frac{\partial^2 E_e}{\partial \mathbf{x}_i^2} & \frac{\partial^2 E_e}{\partial \mathbf{x}_i \partial \mathbf{x}_j} \\ \frac{\partial^2 E_e}{\partial \mathbf{x}_i \partial \mathbf{x}_j} & \frac{\partial^2 E_e}{\partial \mathbf{x}_j^2} \end{bmatrix} = \sum_{e=\{i,j\}} \begin{bmatrix} \mathbf{H}_e & -\mathbf{H}_e \\ -\mathbf{H}_e & \mathbf{H}_e \end{bmatrix}$$

Figure 2.9: def of H

$$\nabla^2 g = \frac{1}{h^2}\mathbf{M} + \mathbf{H}$$

Now, we can use Newton's method for optimization:

$$\mathbf{x}^{n+1} = \mathbf{x}^n - \mathbf{H}^{-1}\nabla g(\mathbf{x}^n)$$

Here, it actually involves a part of Line Search. The general process for optimizing a problem based on line search methods is: 1. Determine the search direction $\mathbf{p}$ (here, $\mathbf{p} = \mathbf{H}^{-1}\nabla g$), 2. Then determine the step size α to advance (this step is called Line Search), 3. Finally, update $\mathbf{x}^{n+1} = \mathbf{x}^n - \alpha\mathbf{p}$. Since the recommended step size for Newton's method is 1, we will not perform additional Line Search here.

$\mathbf{H}$ is a sparse matrix (only neighboring vertices will have corresponding non-zero elements in the matrix), and we use 'Eigen::SparseMatrix' to store it.

If we want to fix points during the solving process, simply setting the fixed points back to their original positions afterward may lead to excessive stretching of the boundary region. The root of the problem lies in considering hard constraints during the solving process.

We can formulate the problem as a constrained optimization problem:

$$\min_{\mathbf{x}} \quad g(\mathbf{x}) = \frac{1}{2h^2}(\mathbf{x}-\mathbf{y})^\top \mathbf{M}(\mathbf{x}-\mathbf{y}) + E(\mathbf{x})\text{s.t.} \quad c(\mathbf{x}) = \mathbf{S}\mathbf{x} = \mathbf{0}$$

In a broad sense, this is indeed a constrained optimization problem. However, we don't necessarily need to use the method of Lagrange multipliers to solve it.

Instead, we can follow the approach used in Homework 3 for handling boundary conditions: modify the Hessian matrix while solving the equations, setting the coefficients corresponding to the fixed points to 1. Alternatively, we can use the transformation $\mathbf{H}^{\text{new}} = \mathbf{S}^T\mathbf{H}\mathbf{S}$ to obtain a smaller matrix, where S is the selection matrix.

2.7.2.3 Interactions

Simulating collision and friction is also an important topic in computer graphics. If not handled properly, it can lead to the common ”popping” issue in games.

For this assignment, we can also consider the collision between a spring-mass system and a sphere. We can even make this sphere move.

We can use a simple penalty-based contact force as follows:

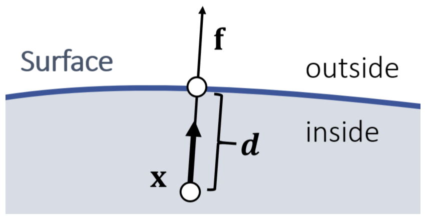


Figure 2.10: Principle of collision

$$\mathbf{f} = k^{\text{penalty}} \max(s \cdot r - \|\mathbf{x} - \mathbf{c}\|, 0) \frac{(\mathbf{x} - \mathbf{c})}{\|\mathbf{x} - \mathbf{c}\|}$$

Here, $\mathbf{c}$ is the center of the sphere, r is the radius of the sphere, s is a magnification factor of the radius (e.g., let $s = 1.1$), which is used to generate contact force even when actual contact has not occurred to reduce visual penetration (denoted as ‘collision_scale_factor‘ in the program), and k^{penalty} is a tunable parameter.

2.8 SPH Fluid

2.8.1 Problem Statement

In this assignment, we will delve into the world of fluids, where you will learn to utilize Smoothed Particle Hydrodynamics (SPH), a classic particle-based simulation method, to compute fluid motion.

Smoothed Particle Hydrodynamics (SPH) is a method for fluid simulation that relies on representing the fluid with a large number of particles and simulating their interactions to mimic fluid motion. The core idea of SPH is to approximate continuous fluid properties (such as density, pressure, velocity, etc.) by summing over discrete particles. This allows SPH to effectively handle large deformations, surface tension, and interactions between fluids and solids.

To use SPH for simulating fluid motion, it is necessary to establish the basic equations of motion for the fluid, typically including equations for mass conservation, momentum conservation, and energy conservation. These equations describe the internal motion of the fluid, as well as interactions between the fluid and solid boundaries or other fluids. In SPH, these equations are discretized based on each particle in the fluid, and then approximated by summing over neighboring particles to approximate the conservation equations for continuous fluid.

In terms of numerical solving, SPH typically involves discretizing the equations in both time and space. Time discretization often employs explicit or implicit time integration methods such as Euler's method, Runge-Kutta method, etc. Spatial discretization involves effectively representing physical quantities like density, pressure, velocity, etc., and updating these quantities based on interactions between neighboring particles.

In practical applications, SPH must account for various complex physical phenomena such as surface tension, viscosity, shock waves in fluid flow, etc. Therefore, selecting appropriate numerical methods and parameter tuning is crucial for obtaining accurate simulation results. Additionally, to enhance simulation efficiency, considerations such as spatial partitioning, particle sorting, etc., are often taken into account.

2.8.2 Propose Algorithms

2.8.2.1 Navier-Stokes Equations

Fluid simulation and the solution of general partial differential equations (PDEs) are fundamentally similar, both based on appropriate temporal and spatial discretization formats, utilizing numerical methods to solve the governing PDEs describing their motion. Within fluid dynamics, we solve the Navier-Stokes equations:

$$\begin{aligned} &\rho\frac{D\mathbf{v}}{Dt} = \rho\mathbf{g} - \nabla p + \mu\nabla^2\mathbf{v} \\ &\nabla\cdot\mathbf{v} = 0 \end{aligned} \tag{2.8.1}$$

where ρ is the fluid density, $\mathbf{v}$ is the velocity field, $\mathbf{g}$ is the gravitational acceleration, p is the pressure within the fluid, and μ is the viscosity coefficient. The right-hand side terms of the first equation correspond to gravity $\rho\mathbf{g}$, pressure $-\nabla p$, and viscous forces $\mu\nabla^2\mathbf{v}$.

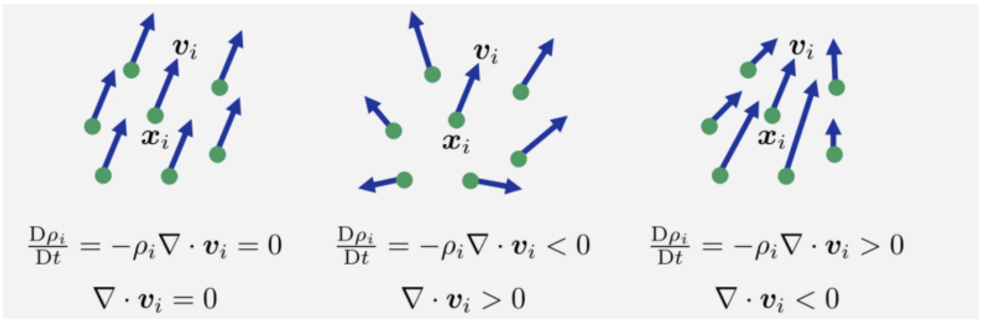
It's worth mentioning that the term $\frac{D\cdot}{Dt}$ in equation (2.8.1) is referred to as the "material derivative" or "derivative following the motion," which is a special case of the total derivative in fluid mechanics:

Consider a fluid element moving with the flow, and at each point within this element, we sample a physical quantity $f = f(\mathbf{x}, t)$, where $\mathbf{x}$ is the spatial position and t is time. As the fluid element moves, $\mathbf{x} = \mathbf{x}(t)$ becomes a function of time. Then, the total derivative of f with respect to time at the sample point is given by: $\frac{df}{dt} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial x}\frac{\partial x}{\partial t} = \frac{\partial f}{\partial t} + \mathbf{v}\cdot\nabla f$. We denote this as the material derivative of the physical quantity f on the moving fluid element, expressed as $\frac{Df}{Dt}$, which accounts for both the change of f due to time and the spatial motion of the sampling point.

NS equation's first line cannot be independently solved. To determine the pressure p, we need to consider the incompressibility condition of the fluid:

$$\nabla \cdot \mathbf{v} = 0 \tag{2.8.2}$$

Combining the physical meaning of divergence (fluid convergence and divergence), we can see that incompressibility is equivalent to mass conservation. That is, the material derivative of the fluid density at any position in space is zero (if the divergence is not equal to zero, i.e., the fluid converges to or diverges from a point, then the mass of the fluid at that point will increase or decrease, violating mass conservation). This is illustrated in Figure 2.11.



$\boldsymbol{v}_i$
$\boldsymbol{x}_i$
$\frac{\mathrm{D}\rho_i}{\mathrm{D}t} = -\rho_i \nabla \cdot \boldsymbol{v}_i = 0$
$\nabla \cdot \boldsymbol{v}_i = 0$
$\frac{\mathrm{D}\rho_i}{\mathrm{D}t} = -\rho_i \nabla \cdot \boldsymbol{v}_i < 0$
$\nabla \cdot \boldsymbol{v}_i > 0$
$\frac{\mathrm{D}\rho_i}{\mathrm{D}t} = -\rho_i \nabla \cdot \boldsymbol{v}_i > 0$
$\nabla \cdot \boldsymbol{v}_i < 0$

Figure 2.11: Illustration of incompressibility condition

2.8.2.2 Spatiotemporal Discretization

In terms of temporal discretization, a common approach in fluid simulation is a technique called ”operator splitting,” which divides the Navier-Stokes equations into two parts:

1. Without considering pressure, updating $\mathbf{v}$:

$$\rho \frac{D\mathbf{v}}{Dt} = \rho\mathbf{g} + \mu\nabla^2\mathbf{v}$$

2. Considering pressure, calculating pressure, and updating $\mathbf{v}$:

$$\rho \frac{D\mathbf{v}}{Dt} = -\nabla p$$

3. Based on the final velocity, updating particle positions.

In terms of spatial discretization, Smoothed Particle Hydrodynamics (SPH) samples physical fields using particles and approximates the physical quantities at unsampled positions through kernel functions. Commonly used kernel functions in SPH are as follows (d is the dimension of the simulation):

$$W(r,h) = \sigma_d \begin{cases} 6(q^3 - q^2) + 1 & \text{for } 0 \le q \le \frac{1}{2} \\ 2(1-q)^3 & \text{for } \frac{1}{2} \le q \le 1 \\ 0 & \text{otherwise} \end{cases}$$
$$\text{with } q = \frac{1}{h}\|r\|, \sigma_1 = \frac{4}{3h}, \sigma_2 = \frac{40}{7\pi h^2}, \sigma_3 = \frac{8}{\pi h^3}$$

2.8.2.3 Computational Formula

The computational formulas in SPH for density and velocity divergence are as follows:

1. Density calculation:

$$\rho_i = \sum_{j+i} \left(\frac{m_j}{\rho_j}\right) m_j W(\mathbf{x}_i - \mathbf{x}_j, h) = \sum_{j+i} m_j W_{ij} \tag{2.8.3}$$

Here, we denote $W_{ij} = W(\mathbf{x}_i, \mathbf{x}_j, h)$, where j represents all neighboring particles of particle i. In SPH, h represents the radius of the kernel function, and we use Δt to represent the time step.

Note that when calculating the density of a particle, we need to consider its own contribution to density, $m_i W(0, h)$ (using the function `W_zero`).

In the program, we assume that each fluid particle has the same mass (equal to the volume multiplied by the density of the fluid at rest, where the resting density is 1000, the particle radius is set to 0.025, and the particle volume is estimated using a cube; please refer to the implementation of the `ParticleSystem` class), which can be accessed using `ps_.mass()`.

2. Velocity divergence calculation:

$$\nabla \cdot \mathbf{v}_i = \sum_j \frac{m_j}{\rho_j} (\mathbf{v}_j - \mathbf{v}_i) \cdot \nabla W_{ij} \tag{2.8.4}$$

For the calculation of viscosity, we recommend using the formula:

$$\nabla^2 \mathbf{v}_i = 2(d+2) \sum_j \frac{m_j}{\rho_j} \frac{\mathbf{v}_{ij} \cdot \mathbf{x}_{ij}}{\|\mathbf{x}_{ij}\|^2 + 0.01h^2} \nabla W_{ij} \tag{2.8.5}$$

Where d is the dimension of the simulation, which is 3 in this case. We divide the viscosity coefficient μ by the particle density to obtain the value $\nu = \frac{\mu}{\rho}$ (referred to as the kinematic viscosity coefficient), which serves as a tunable simulation parameter named `viscosity`.

3. pressure calculation:

The task for this assignment is to implement a classical method called ”Weakly Compressible Smoothed Particle Hydrodynamics” (WCSPH).

The pressure calculation follows the equation of state (EOS) formula:

$$p_i = k_1 \left(\left(\frac{\rho_i}{\rho_0} \right)^{k_2} - 1 \right) \tag{2.8.6}$$

In the program, k_1 is represented by the parameter `stiffness`, and k_2 is represented by the parameter `exponent`. Note that the pressure should generally be greater than 0. In case of insufficient particles, you can set $p_i = \max(0.0, p_i)$.

The acceleration due to pressure is given by $-\frac{1}{\rho}\nabla p$, where:

$$\nabla p_i = \rho_i \sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) \nabla W_{ij} \tag{2.8.7}$$

Here, ∇p_i represents the gradient of pressure for particle i, ρ denotes the density, m is the mass, and W_{ij} is the kernel function.

2.9 Character Animation

2.9.1 Problem Statement

In this task, we are implementing character animation. Although there is a lot of research in this area, our task is to achieve the simplest results. To simulate a character, we first need to study the changes in the skeleton and then address the skin. The skin is related to certain points on the skeleton and is handled using a method similar to interpolation. For the skeleton, we only need to consider the transformation of coordinates.

Skeleton animation is the foundation of character animation, primarily involving the transformation of the skeleton's coordinates. The skeleton typically consists of a series of joints connected by bones, forming a hierarchical structure. Each joint has a position and a rotation, and through these transformations, various character movements can be realized. We only need to focus on the transformations of the joints, specifically the changes in their positions and rotations over time.

Based on the changes in the skeleton, we then need to study the skin. Skin deformation applies the skeleton's animations to the character's mesh, allowing the character's appearance to change with the movement of the skeleton. Each point on the skin is influenced by certain points on the skeleton, and we use interpolation to handle the deformation of these points. Each skin point is affected by multiple joints, and the final deformation position is calculated based on weights.

By transforming the skeleton's coordinates and interpolating the skin points, we can achieve basic character animation. Although this method is simple, it can produce relatively natural character movements and provides a foundation for more complex animations in the future.

2.9.2 Propose Algorithms

2.9.2.1 Skeletal Animation

Skeleton animation consists of two main components:

1. Skeleton: A low-degree-of-freedom hinge system that acts as the driver.
2. Skinning: The surface mesh bound to the skeleton. The movement of the mesh vertices is influenced by multiple skeleton joints, and their final spatial coordinates are calculated through interpolation and other methods.

Due to its wide applicability and efficiency, skeletal animation is widely used in games and animations.

We can observe that the movement of parent joints will drive the movement of child joints, forming a tree-like structure similar to the diagram below:

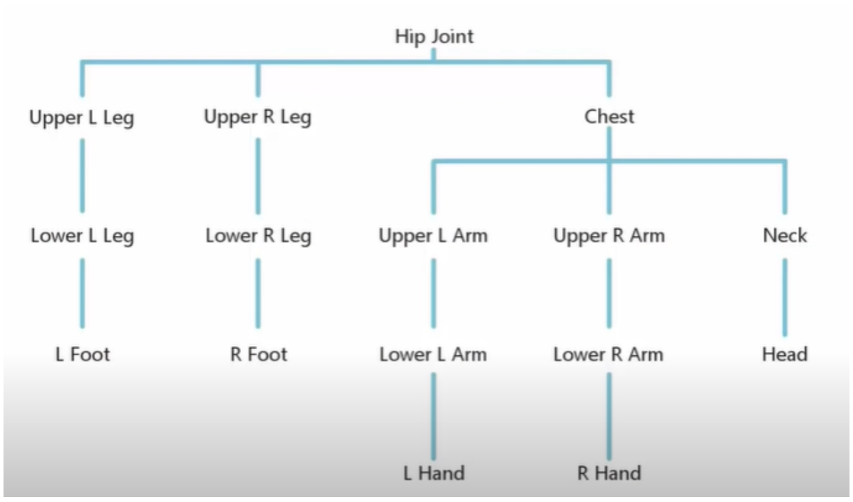


Figure 2.12: Illustration of Skeleton

Next, let's introduce some important transformation matrices:

1. **localTransform**: The coordinate transformation of the joint in the local coordinate system (4x4 matrix).
2. **worldTransform** (or global transform): The coordinate transformation of the joint in the world coordinate system (4x4 matrix). For the root joint, in our assignment, we can directly set its $worldTransform = localTransform$.
3. **bindTransform**: The position and orientation of the joint relative to the mesh vertices in the bind pose (e.g., T-pose). This is typically defined during the creation of the 3D model and remains constant throughout the animation's lifecycle. Its role will be seen in the skinning section below.

For a child joint, its *worldTransform* is the composite of the parent joint's *worldTransform* and the child joint's *localTransform*:

$$\text{worldTransform}_{\text{child}} = \text{worldTransform}_{\text{parent}} \times \text{localTransform}_{\text{child}}$$

This hierarchical transformation ensures that changes in a parent joint's position and orientation will correctly propagate to its child joints, maintaining the integrity of the skeletal structure during animation.

2.9.2.2 Skinning Animation

In skeletal animation, skinning refers to the process of deforming a character's mesh based on the transformations of its skeleton. This ensures that the surface mesh moves naturally with the underlying bone structure.

One of the most common techniques for skinning is Linear Blend Skinning (LBS), also known as smooth skinning. In LBS, each vertex of the mesh is influenced by multiple joints, and its final position is calculated based on the weighted transformations of these joints.

1. **Vertex Weights**: Each vertex v_i on the mesh is associated with one or more joints, with each association having a corresponding weight w_{ij}. These weights determine the influence each joint has on the vertex. The sum of weights for each vertex is typically normalized to 1.
2. **Vertex Transformation**: The final position v_i' of a vertex v_i in the animated mesh is calculated as the weighted sum of the transformed positions influenced by each joint. Mathematically, this is expressed as:
$$v_i' = \sum_j w_{ij} \cdot (\text{worldTransform}_{J_j} \cdot \text{bindTransform}_{J_j}^{-1} \cdot v_i)$$
Here, $\text{bindTransform}_{J_j}$ is the bind pose transformation of joint J_j, and $\text{worldTransform}_{J_j}$ is the current world transformation of joint J_j.

Chapter 3 Modeling Course in USTC

This chapter comes from Modeling course in USTC, 2024, taught by Renjie Chen.

The homepage is listed: `http://staff.ustc.edu.cn/~renjiec/mm2024/`

3.1 Seam Carving for Content-Aware Image Resizing

3.1.1 Problem Statement

We need to adjust the size of the given image, and this adjustment is not very straightforward. In this project, we only consider resizing the image, so the core issue is to consider the removal of certain pixels. To achieve this, we propose an energy function for each pixel. Naturally, we aim to remove pixels with lower energy, as they are relatively less important in the image. Additionally, we need to maintain the continuity of the image to prevent significant information distortion. Therefore, we introduce the concept of seams.

3.1.2 Propose Algorithms

3.1.2.1 The Energy Function

We use the common sum of gradients energy function.

$$e_1(I) = |\frac{\partial I}{\partial x}| + |\frac{\partial I}{\partial y}| \tag{3.1.1}$$

The MATLAB has provided us the energy function.

```
costfunction = @(im) sum(imfilter(im, [.5 1 .5; 1 -6 1; .5 1 .5]).^2, 3);
```

This MATLAB expression defines a cost function named costfunction, which takes an image im as input. Within the function, the imfilter operation applies a specific filter to the input image, which is designed to detect edges. The filter used here is a 3x3 convolution kernel with weights [.5 1 .5; 1 -6 1; .5 1 .5]. After applying the filter, the result is squared element-wise and then summed along the third dimension (which typically represents color channels in an RGB image) using the sum function. This operation computes the squared magnitude of the filtered image, effectively emphasizing the presence of edges in the image. Thus, the costfunction evaluates the overall edge strength in the input image.

3.1.2.2 Seams

Formally, let I be an n×m image and define a vertical seam to be:

$$\mathbf{s^x} = \{s_i^x\}_{i=1}^n = \{(x(i), i)\}_{i=1}^n \tag{3.1.2}$$

In equation 3.1.2,

$$\forall\, i,\ |x(i) - x(i-1)| \leq 1.$$

where x is a mapping $x : [1, ..., n] \to [1, ..., m]$. That is, a vertical seam is an 8-connected path of pixels in the image from top to bottom, containing one, and only one, pixel in each row of the image.

Similarly, if y is a mapping $y : [1, ..., m] \to [1, ..., n]$, then a horizontal seam is:

$$\mathbf{s}^{\mathbf{y}} = \{s_i^y\}_{i=1}^n = \{(y(i), i)\}_{i=1}^n \tag{3.1.3}$$

In equation 3.1.3,

$$\forall\, i,\ |y(i) - y(i-1)| \leq 1.$$

3.1.2.3 Optimization

We look for the optimal seam s^* that minimizes the seam cost :

$$s^* = \min_s E(s) = \min_s \sum_{i=1}^n e(I(s_i)). \tag{3.1.4}$$

The optimal seam can be found using dynamic programming. The first step is to traverse the image from the second row to the last row and compute the cumulative minimum energy M for all possible connected seams for each entry (i, j):

$$M(i,j) = e(i,j) + \min(M(i-1, j-1), M(i-1, j), M(i-1, j+1)). \tag{3.1.5}$$

At the end of this process, the minimum value of the last row in M will indicate the end of the minimal connected vertical seam. Hence, in the second step we backtrack from this minimum entry on M to find the path of the optimal seam. The definition of M for horizontal seams is similar.

3.1.2.4 Resizing

The project just requires us to delete the columns. We also provide an easy way to delete the rows. In the paper, the author uses another optimization, leading to a better outcome.

MATLAB has provided a function imrotate.

```
rim = imrotate(im, 90)
```

In this way, we can rotate the image. If we choose to rotate it by 90 angle, the vertical direction can be changed into horizon direction, Then we can use the deleting function for vertical direction.

3.2 Image Processing

3.2.1 Problem Statement

We have two small tasks in image processing: image compression and low-rank image restoration. To achieve this, we primarily utilize Singular Value Decomposition (SVD). In image processing, an image can be represented as a matrix, where each element represents the grayscale value or color component of a pixel. Singular Value Decomposition is a method of decomposing a matrix into the product of three matrices: an orthogonal matrix, a diagonal matrix, and the transpose of another orthogonal matrix. In this decomposition, the diagonal matrix contains singular values, where larger singular values correspond to more information in the image. Image compression mainly involves preserving the larger singular values, while low-rank restoration involves decomposing the matrix into a low-rank component and noise.

3.2.2 Propose Algorithms

3.2.2.1 Image Compression

Image compression is the main algorithm we are implementing in this project. First, we need to read information from the image. In image processing, an image can be viewed as a matrix, where each element represents the pixel's grayscale value or color component. MATLAB implements this information through the following function:

```
% Read the image
im = imread('image.jpg');

% Get the image dimensions
[rows, cols, channels] = size(image);
```

The variable im contains the image information and allows access to the pixels of each channel. By fixing a channel, we can obtain a color pixel matrix corresponding to each point. For such matrices, we can directly use the SVD decomposition, which is already provided as a function in MATLAB.

```
% SVD method
[U,S,V] = svd(A);
```

In this equation,

1. $\boldsymbol{A}$ is the input matrix.
2. $\boldsymbol{U}$ is an m-by-m unitary matrix representing the left singular vectors.
3. $\boldsymbol{S}$ is an m-by-n diagonal matrix containing the singular values of $\boldsymbol{A}$.
4. $\boldsymbol{V}$ is an n-by-n unitary matrix representing the right singular vectors.

We truncate using the following approach: Since users typically don't know the rank of the matrix, we include the rank of the matrix for evaluation. If the specified retention value is greater than the rank, we directly return the original matrix. However, if the retention value is smaller, we set the elements beyond the retention value to zero, as the singular values provided by the library function are already sorted in descending order.

```
rank_S = rank(S);
% Change the color information
if (k >= rank_S)
    im(:,:,i) = channel;
else
    for j = k+1:rank_S
```

```
        S(j,j) = 0;
    end
    im(:,:,i) = U*S*V';
end
```

Surprisingly, V' directly implements the transpose of V, making the notation quite concise.

"I didn't find a function in the existing C++ library that provides image information as conveniently as in MATLAB. Instead, I found the 'get_pixel' function in the IMGUI library to obtain image information. For each channel stored in Eigen library matrices, I constructed the information of each matrix into another dynamic array.

```
for (int i = 0; i < data_->width(); ++i)
    {
        for (int j = 0; j < data_->height(); ++j)
        {
            A[ch](i,j) = static_cast<double>(data_->get_pixel(i, j)[ch]);
        }
    }
```

We use the Eigen library for SVD decomposition. However, C++ provides a direct truncation function. The code is as follows, and the logic is identical to MATLAB:

```
int num = static_cast<int>(singularValues.size());
    if (k >= num)
    {
        continue;
    }
    else
    {
        A[ch] = U.leftCols(k)* singularValues.head(k).asDiagonal() * V.leftCols(k).
            transpose();
    }
```

In the end, we use the set_pixel function to assign colors to each pixel.

```
std::vector<uchar> col;
for (int ch = 0; ch < 3; ch++)
{
     col.push_back(static_cast<uchar>(std::clamp(A[ch](i,j), 0., 255.)));
}
data_->set_pixel(i, j, col);
```

For grayscale image conversion, we apply the same logic, but there are some changes in color assignment.

```
uchar gray_value = (color[0] + color[1] + color[2]) / 3;
data_->set_pixel(i, j, { gray_value, gray_value, gray_value });
```

3.2.2.2 Low-rank Image Restoration

This is another task for this assignment. Here, only the most basic approach has been used. Also, a fully functional interactive system has not been implemented.

For some highly symmetric images, it's evident that pixels in different rows or columns are very close to each other. Thus, the pixel matrix has a low rank. However, certain noise can lead to abnormal increases in the rank of the image matrix. We use the Robust PCA method to decompose a matrix into a low-rank component and a noise component, and then draw them separately, achieving a restoration effect.

Robust PCA (Principal Component Analysis) is a technique used for matrix decomposition, particularly for dealing with data that contains outliers or noise. It aims to separate a given matrix into two components: a low-rank matrix representing the underlying structure or signal, and a sparse matrix representing the noise or outliers. The algorithm minimizes the following objective function:

$$\min_{L,S} \|L\|_* + \lambda \|S\|_1 \tag{3.2.1}$$

where:

1. L is the low-rank matrix representing the underlying structure.
2. S is the sparse matrix representing the noise or outliers.
3. $\|L\|_*$ denotes the nuclear norm of L, which is the sum of its singular values and serves as a proxy for its rank.
4. $\|S\|_1$ denotes the L_1-norm of S, promoting sparsity in S.
5. λ is a parameter that balances between the low-rank and sparse components.

The algorithm iteratively updates L and S until convergence, typically using optimization techniques such as alternating direction method of multipliers (ADMM) or proximal gradient descent.

Robust PCA has various applications in computer vision, image processing, and data analysis, where it can be used for tasks such as background subtraction, denoising, and anomaly detection. It provides a robust way to decompose data into meaningful components, even in the presence of outliers or noise.

Specifically, in this experiment, we adopt the ADMM (Alternating Direction Method of Multipliers) Robust PCA approach. The ADMM algorithm iteratively updates L, S, and a Lagrange multiplier Y as follows:

1. Update L (matrix completion step):

$$L = \text{SVD\_Threshold}(M - S + \frac{1}{\rho} Y) \tag{3.2.2}$$

2. Update S (sparse recovery step):
$$S = \text{svt}(M - L + \frac{1}{\rho}Y, \frac{\lambda}{\rho}) \tag{3.2.3}$$
3. Update Lagrange multiplier Y:
$$Y = Y + \rho(M - L - S) \tag{3.2.4}$$
4. Compute primal and dual residuals:
$$\text{Primal\_Residual} = \|M - L - S\|_F \tag{3.2.5}$$
$$\text{Dual\_Residual} = \rho\|L - L_{\text{old}}\|_F \tag{3.2.6}$$
5. Check convergence: If Primal_Residual < Tolerance and Dual_Residual < Tolerance, then break loop
6. Store previous low-rank component:
$$L_{\text{old}} = L \tag{3.2.7}$$

Specifically, SVD_Threshold is related to SVD method, svt is a singular value thresholding operator.

3.3 Interpolation and Neural Work

3.3.1 Problem Statement

The tasks we are undertaking this time are related to interpolation. The first task involves inputting points on a canvas and using various interpolation methods and parameterization techniques to draw curves determined by these points, including fitting methods for model training. The second task involves using a neural network to classify insect data. We can choose different neural network models and activation functions, and we can also adjust some parameters.

3.3.2 Propose Algorithms

3.3.2.1 Curve Reconstruction

For our task, we aim to fit curves by inputting points on a canvas and employing various parameterization methods and interpolation techniques. We have implemented four different parameterization methods and five different interpolation methods.To use our demo, you first need to select an interpolation function from the first row and then choose a parameterization method from the second row. This will allow you to display the desired results.

Our core approach is to first allow users to input points via left-clicking on the interface, as illustrated in Figure 3.1. The input points will simultaneously be recorded in a file named points.txt. Subsequently, we will proceed with parameterizing the curves. The ”clear” button indicates clearing all current drawings on the graph as well as removing all points stored in the ”points” file.

During parameterization, the first and last input points represent states at time 0 and 1, respectively.

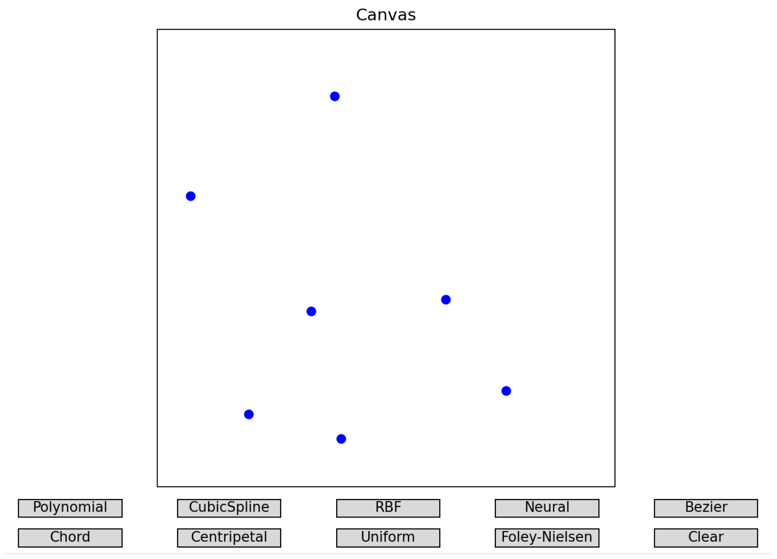


Figure 3.1: GUI for Curve Reconstruction

This approach allows us to normalize the parameterization after the process. The mathematical expression for parameterization is as follows:

1. Chord Parameterization:

$$t_i = \frac{\sum_{j=1}^{i} \sqrt{(x_j - x_{j-1})^2 + (y_j - y_{j-1})^2}}{\max\left(\sum_{j=1}^{n} \sqrt{(x_j - x_{j-1})^2 + (y_j - y_{j-1})^2}\right)}.$$

2. Centripetal Parameterization:

$$t_i = \frac{\sum_{j=1}^{i} \sqrt{\sqrt{(x_j - x_{j-1})^2 + (y_j - y_{j-1})^2}}}{\max\left(\sum_{j=1}^{n} \sqrt{\sqrt{(x_j - x_{j-1})^2 + (y_j - y_{j-1})^2}}\right)}.$$

3. Uniform Parameterization:

$$t_i = \frac{i}{n}.$$

4. Foley-Nielsen Parameterization:

$$\begin{aligned} t_0 &= 0 \\ t_i - t_{i-1} &= \sum_{i=1}^{n} N(P_i, P_{i-1}) \\ N(P_i, P_{i-1}) &= \begin{cases} d_1\left(1 + \frac{3}{2}\frac{\hat{\alpha}_1 d_1}{d_1+d_2}\right) & \text{if } i = 1 \\ d_i\left(1 + \frac{3}{2}\frac{\hat{\alpha}_{i-1} d_{i-1}}{d_{i-1}+d_i} + \frac{3}{2}\frac{\hat{\alpha}_i d_i}{d_i+d_{i+1}}\right) & \text{if } i = 2, \ldots, n-1 \\ d_n\left(1 + \frac{3}{2}\frac{\hat{\alpha}_{n-1} d_{n-1}}{d_{n-1}+d_n}\right) & \text{if } i = n \end{cases} \\ \text{where,} \quad d_i &= ||P_i - P_{i-1}||_2, \\ \alpha_i &= \text{angle}(P_{i-1}, P_i, P_{i+1}), \\ \hat{\alpha}_i &= \min(\pi - \alpha_i, \frac{\pi}{2}) \end{aligned}$$

We have utilized five different interpolation functions, implemented through libraries, for our task.

1. Polynomial Interpolation: Polynomial interpolation fits a polynomial function to the given data points. It aims to find a polynomial of degree n that passes exactly through $n + 1$ data points. This method is straightforward and widely used due to its simplicity.
2. Cubic Spline Interpolation: Cubic spline interpolation divides the curve into smaller segments and fits a cubic polynomial to each segment. It maintains smoothness by enforcing continuity of derivatives between adjacent segments, resulting in a piecewise smooth curve.
3. Radial Basis Function (RBF) Interpolation: RBF interpolation constructs an interpolant by fitting a sum of radial basis functions to the given data points. Each basis function depends only on the distance from the center of a chosen point, enabling the interpolation of scattered data.
4. Neural Network Interpolation: Neural network interpolation utilizes a neural network model to approximate the relationship between input and output data points. The neural network learns the mapping between input and output data through training, allowing it to predict values at unobserved points.
5. Bézier Interpolation: Bézier interpolation constructs a curve defined by control points and interpolation of these points to generate a smooth curve. It is commonly used in computer graphics and CAD software for its simplicity and ability to produce aesthetically pleasing curves.

Once parameterized and interpolated, we can reconstruct the curve of the given input points.

3.3.2.2 Insect Classification Models

The core objective of this task is to train a neural network model using labeled data that has already been identified. Subsequently, the trained model can be used to predict outcomes for unknown data.

First, we need to determine the target of model learning, which is typically the given data multiplied by the learning rate, set to 0.8. Then, we need to establish the model. In this project, we have built two different types of neural network models: MLP and RNN. For the MLP model, we have utilized three different activation functions, which can be adjusted in the code. Unused code segments need to be commented out. Finally, we evaluate the accuracy of the model and visualize the evaluation, including distribution visualization and the changes of train loss and accuracy over training epochs.

Model training typically requires the following parameters:

1. input size: This parameter represents the number of features in the input data. In this context, it indicates the dimensionality of the input data fed into the neural network.
2. hidden size: It refers to the number of neurons or units in the hidden layer of the neural network. This parameter controls the complexity and capacity of the model to learn and represent patterns in the data.
3. num classes: This parameter specifies the number of classes or categories in the classification task. It defines the output dimensionality of the neural network, indicating the number of distinct labels the model is trained to predict.

4. learning rate: The learning rate is a hyperparameter that controls the step size of the optimization algorithm during training. It determines how much the model parameters are adjusted in response to the gradient of the loss function. A higher learning rate can lead to faster convergence but may risk overshooting the optimal solution, while a lower learning rate may result in slower convergence but more stable training.
5. num epochs: This parameter represents the number of training epochs, which is the number of times the entire dataset is passed forward and backward through the neural network during training. Each epoch consists of one forward pass to compute the loss and one backward pass to update the model parameters based on the computed gradients. Increasing the number of epochs allows the model to see the data multiple times and learn from it more thoroughly, potentially improving performance.

Indeed, adjusting the learning rate and the number of epochs can have a significant impact on the training process and the resulting model performance.

1. Learning Rate (lr):
   (a). Higher learning rates can lead to faster convergence but risk overshooting the optimal solution.
   (b). Lower learning rates may result in slower convergence but offer more stability and smoother training dynamics.
   (c). It's crucial to find an appropriate learning rate that balances convergence speed and stability to achieve optimal performance.
2. Number of Epochs (num epochs):
   (a). Increasing the number of epochs allows the model to see the data more times, potentially improving performance.
   (b). However, excessive epochs may lead to overfitting if the model memorizes the training data without generalizing well to unseen data.
   (c). Regularization techniques such as early stopping or dropout can help mitigate overfitting and determine the optimal number of epochs.

In this experiment, we employed two distinct neural network architectures.

1. MLP (Multi-Layer Perceptron): MLPs are feedforward neural networks with multiple layers of interconnected neurons. They excel at learning complex patterns in static input-output mappings, making them suitable for tasks such as classification and regression. With fully connected layers and nonlinear activation functions, MLPs can approximate any continuous function to arbitrary accuracy given enough parameters. However, they lack memory and context, which limits their effectiveness in processing sequential data or data with temporal dependencies.
2. RNN (Recurrent Neural Network): RNNs are specialized for sequential data with temporal dependencies. Unlike MLPs, RNNs have recurrent connections that allow information to persist over time. This enables them to capture temporal dynamics and long-range dependencies in sequential data, making them well-suited for tasks such as time series prediction, natural language processing, and speech recognition. However, RNNs are susceptible to the vanishing

gradient problem, where gradients diminish exponentially as they propagate backward through time, hindering their ability to learn long-term dependencies effectively.

If the data is static, then training with MLP is a reasonable choice, while the other model can serve as a point of comparison. This allows for a comparison of the performance of different models on the same dataset, thereby providing a more comprehensive evaluation of their effectiveness and suitability.

In addition, we used three different common activation functions in the MLP. The mathematical formulas are as follows:

1. Rectified Linear Unit (ReLU) Activation Function:
$$f(x) = \max(0, x)$$
2. Sigmoid Activation Function:
$$f(x) = \frac{1}{1 + e^{-x}}$$
3. Hyperbolic Tangent (tanh) Activation Function:
$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Here's a brief description of the characteristics of the three activation functions:

1. Rectified Linear Unit (ReLU) Activation Function: ReLU is a simple and efficient activation function that outputs zero for negative input values and linearly increases for positive input values. It overcomes the vanishing gradient problem and accelerates convergence by enabling faster and more effective learning in deep neural networks. However, ReLU neurons can suffer from the "dying ReLU" problem, where they become inactive and output zero for all inputs, leading to dead neurons in the network.
2. Sigmoid Activation Function: The sigmoid function squashes the input values between 0 and 1, making it suitable for binary classification tasks. However, it suffers from the vanishing gradient problem, particularly for inputs far from zero, which can slow down learning in deep neural networks.
3. Hyperbolic Tangent (tanh) Activation Function: Similar to the sigmoid function, the tanh function squashes the input values, but between -1 and 1. It provides stronger gradients compared to the sigmoid, making it more suitable for training deep neural networks. However, like the sigmoid, it also suffers from the vanishing gradient problem.

Chapter 4 GAMES102: Geometry Modeling And Processing

This chapter comes from GAMES102 Course, taught by Ligang Liu.

The homepage is listed: `http://staff.ustc.edu.cn/~lgliu/Courses/GAMES102_2020/default.html`

4.1 Data Fitting

The main purpose is to find a function which fits the data. The input is some observed data, and the output should be a funtion which can react to the information. We need to find:

1. function space,
2. which function in the space,
3. how to determine the function.