UNIVERSIDADE FEDERAL DO ABC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

Candy Veronica Tenorio Gonzales

Marmoreio Digital com Interface Natural

Santo André - SP
2017

Candy Veronica Tenorio Gonzales

Marmoreio Digital com Interface Natural

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal do ABC, como requisito parcial à obtenção do título de Mestre em Ciência da Computação. Linha de Pesquisa: Computação Científica e Aplicada.

Orientador: Prof° Dr. Mario Alexandre Gazziro

Coorientador: Prof° Dr. João Paulo Gois

Santo André - SP

2017

Sistema de Bibliotecas da Universidade Federal do ABC
Elaborada pelo Sistema de Geração de Ficha Catalográfica da UFABC
com os dados fornecidos pelo(a) autor(a).

Tenorio Gonzales, Candy Veronica
Marmoreio Digital com Interface Natural / Candy Veronica Tenorio Gonzales. — 2017.

79 fls. : il.

Orientador: Mario Alexandre Gazziro

Dissertação (Mestrado) — Universidade Federal do ABC, Programa de Pós Graduação em Ciência da Computação, Santo André, 2017.

1. Marmoreio Digital. 2. GLSL. 3. GPU. 4. HCI. I. Gazziro, Mario Alexandre. II. Programa de Pós-Graduação em Ciência da Computação, 2017. III. Título.

Este exemplar foi revisado e alterado em relação à versão original, de acordo com as observações levantadas pela banca no dia da defesa, sob responsabilidade única do autor e com a anuência de seu orientador.

Santo André, 22 **de** Maio **de 2017.**

Assinatura do autor: _____

Assinatura do orientador: _____

MINISTÉRIO DA EDUCAÇÃO
Fundação Universidade Federal do ABC
Programa de Pós-Graduação em Ciência da Computação
Avenida dos Estados, 5001 – Bairro Santa Terezinha – Santo André – SP
CEP 09210-580 · Fone: (11) 4996-0017
poscomp@ufabc.edu.br

FOLHA DE ASSINATURAS

Assinaturas dos membros da Banca Examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado da candidata Candy Veronica Tenorio Gonzales, realizada em 22 de fevereiro de 2017:

Prof.(a) Dr.(a) **Mario Alexandre Gazziro** (Universidade Federal do ABC) – Presidente

Prof.(a) Dr.(a) **Harlen Costa Batagelo** (Universidade Federal do ABC) – Membro Titular

Prof.(a) Dr.(a) **Paulo Aristarco Pagliosa** (Universidade Federal de Mato Grosso do Sul) – Membro Titular

Prof.(a) Dr.(a) **André Luiz Brandão** (Universidade Federal do ABC) – Membro Suplente

Prof.(a) Dr.(a) **Afonso Paiva Neto** (Universidade de São Paulo) – Membro Suplente

Agradecimentos

Primeiramente agradeço a Deus, que sempre me conduz vitoriosamente em Cristo. Por cuidar da minha vida e por colocar pessoas maravilhosas no meu caminho.

Agradeço eternamente aos meus pais, Jorge e Dora, por seu amor, confiança e educação que tem me dado até hoje. Por não medir esforços para a realização deste sonho. Aos meus irmãos Shirley, George e Nhara pelo apoio e carinho apesar da distancia. Eu sempre estarei agradecida com vocês.

Meus sinceros agradecimentos também aos meus orientadores, os professores e amigos, Mario Gazziro e Jõao Paulo Gois por todos os conselhos, pela paciência e ajuda que necessitei durante esse período de mestrado. Obrigada por seu acompanhamento a este trabalho, pois foram essenciais para a conclusão deste mestrado.

Ao meu time de vôlei feminino da UFABC, por me dar tantas alegrias, vitórias e sobretudo por me ensinar quanto amor tem em Brasil para os estrangeiros. Meu muito obrigada a todos vocês.

Às minhas amigas-irmãs Guisella e Lizbeth por cada mensagem, ligação e viagem em cada momento que eu precisei de vocês. Obrigada por me acompanhar neste sonho.

A CAPES pelo apoio financeiro concedido para realização deste trabalho de pesquisa e assim conseguir a finalização com sucesso.

Finalmente, a minha querida UFABC tanto pelo apoio financeiro quanto pelos conhecimentos fornecidos nas disciplinas cursadas durante o mestrado.

Resumo

Marmoreio em papel é um método de *design* principalmente utilizado por artistas desenvolvedores de capas de livros, no qual tinta a base de óleo é depositada sobre uma superfície aquosa, de forma a produzir padrões texturais semelhantes à do mármore. Os padrões são o resultado da flutuação dos pigmentos de tinta sobre a água definidos pelos movimentos manuais do artista que, em seguida, é cuidadosamente transferido para uma superfície de papel ou tecido absorvente. Apesar do fato de alguns sistemas digitais de marmoreio já terem sido desenvolvidos no passado, nenhum deles pôde fornecer simultaneamente uma interface natural ao usuário e uma simulação em tempo real. Este trabalho obteve a melhora do processo de marmoreio digital através de uma interface fácil de aprender, apresentando simultaneamente características de resposta rápida e de fácil navegação através de uma caneta digital, usando uma mesa digitalizadora como superfície de *design*, processando algoritmos na GPU por meio de programas *shaders* para os cálculos de dinâmica computacional de fluidos, envolvendo resolução em tempo real das Equações de *Navier-Stokes*.

Palavras-chaves: Marmoreio Digital, GPU, *Shaders*, Mesa Digitalizadora.

Abstract

Paper marbling is a design method mainly used by artists who develop book covers, in which is deposited oil-based paint on an aqueous surface in order to produce textured patterns like marble. The patterns are the result of the fluctuation of the ink pigments on the water defined by the artist's manual movements which is then carefully transferred to a surface of absorbent paper or fabric. Although some digital marbling systems have already been developed in the past, none of them could simultaneously provide a natural user interface and real time simulation. This work improvement of the digital marbling process through an easy to learn interface, simultaneously presenting fast response characteristics and easy navigation through of a pen stylus, using a pen tablet as a design surface, processing GPU algorithms using shaders to calculate computational fluids dynamics, involving real time resolution of the Navier-Stokes equations.

Keywords: Digital Marbling, GPU, Shaders, Pen Tablet.

Lista de ilustrações

Lista de tabelas

Lista de siglas

CFD	Computational Fluid Dynamics
Cg	C for Graphics
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FBO	Frame Buffer Object
GLSL	OpenGL Shading Language
GPU	Graphics Processing Unit
HCI	Human-Computer Interaction
MSAA	Multisample anti-aliasing
NUI	Natural User Interface
OpenGL	Open Graphics Library
RAM	Random Access Memory
VBO	Vertex Buffer Object
VRAM	Video Random Access Memory

Sumário

1 Introdução

Marmoreio em papel é um método de *design* decorativo originário da Pérsia, por volta de 1400 D.C., que se expandiu através da Europa (1). Esse método é principalmente utilizado por artistas desenvolvedores de capas de livros, no qual tinta à base de óleo é depositada sobre a superfície da água, de forma a produzir padrões semelhantes à pedra de mármore. Os padrões produzidos, segundo Maurer-Mathison (2), são resultado de um processo que consiste em:

(a) Criar os padrões iniciais: Um líquido de fundo é colocado em uma bandeja, no qual as tintas são dispersas na superfície do líquido com conta-gotas ou pincéis de bambu (Figura 1-(a)) .

(b) Mudar os padrões: Pentes de bambu, agulhas, espátulas com diferentes configurações (Figura 2-(a)) são utilizadas para produzir efeitos mais complexos de marmoreio (Figura 1-(b)).

(c) Capturar os padrões: Uma vez que o desenho está concluído, o artista aplica suavemente uma folha de papel ou tecido ou qualquer outro material absorvente sobre a bandeja, a fim de capturar o resultado (Figura 1-(c)).

Figura 1 – Processo de marmoreio em papel: (a) Criar o padrão inicial, (b) Mudar o padrão inicial, (c) Capturar os padrões.

(a)

(b)

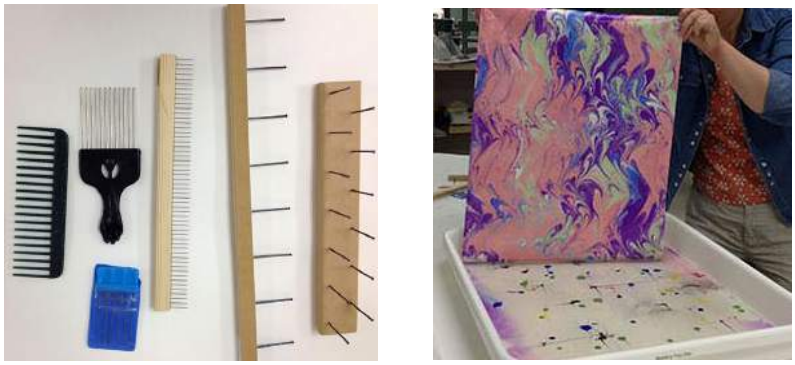
(c)

Fonte – (3)

Como dito anteriormente, para mudar os padrões iniciais a fim de obter um desenho em papel, é necessário o uso de distintas ferramentas. O importante deste uso é que se obtém muitas vezes um único desenho devido à manipulação e à seleção das ferramentas. Assim por exemplo o desenho obtido na Figura 1-(c) é distinto do apresentado na Figura 2-(b).

Figura 2 – Ferramentas de marmoreio em papel: (a) Pentes com diferentes números de dentes, (b) Uso de ferramentas de marmoreio para criação de um padrão em papel.

(a) (b)



Fonte – (4)

A obtenção de padrões em papel através do processo de marmoreio em papel pode ser desenvolvido por computadores, o qual é chamado de *marmoreio digital*. Esse processo tem por objetivo simular os dois primeiros passos do marmoreio em papel, isto é atingido pelo uso de uma interface gráfica de usuário que possa proporcionar uma interação realista. Para criação dos padrões de marmoreio em papel em forma digital é preciso simular o escoamento de um fluido, através de modelagem matemática e computacional de equações. Assim, a simulação de escoamento um fluido pode ser feita de distintas maneiras, sendo as mais populares para sistemas de marmoreio digital:

- A Dinâmica Computacional dos Fluidos 2D (CFD), como nos trabalhos de Mao *et al.* (1) e Jin *et al.* (5);
- O Modelo Matemático, como no trabalho feito por Lu *et al.* (6).

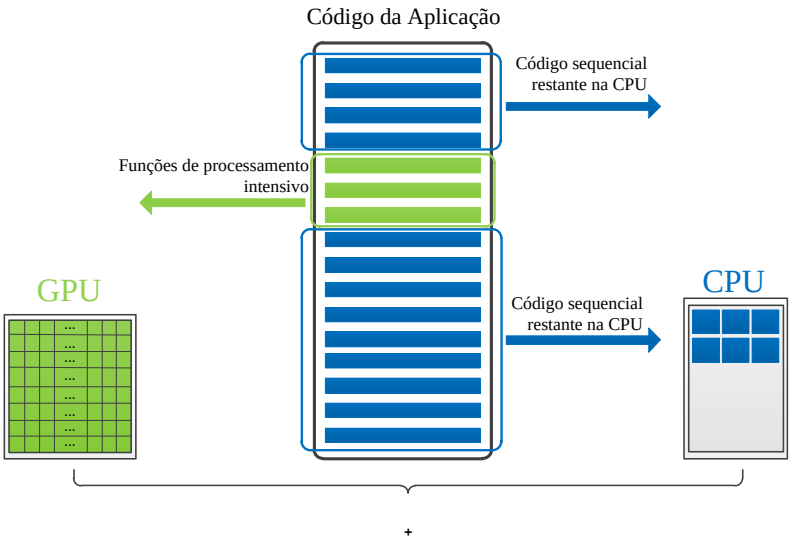
A diferença principal dessas técnicas está baseada em como são realizadas as advecções das tintas (computacionalmente chamaremos as tintas de texturas), ou seja, como essas texturas são criadas e transformadas durante a interação do artista com o sistema de marmoreio digital.

Para garantir um tempo de resposta aceitável para o artista no processo de marmoreio digital, têm-se utilizado na simulação de texturas, um processo de aceleração na unidade de processamento gráfico (do inglês, *Graphics Processing Unit* GPU [1]) como no trabalho feito

[1] É o nome dado a um tipo de microprocessador especializado em processar computacionalmente gráficos mediante processamento paralelo (7).

Zao *et al.* (8). A GPU oferece melhor desempenho em aplicativos (para problemas altamente paralelizáveis) desse modo, um processo computacional paralelizado gerencia um melhor desempenho ao transferir a parte de processamento intensivo à GPU, enquanto o restante é executado de forma sequencial na CPU (Figura 3).

Figura 3 – Distribuição de tarefas e comunicação CPU-GPU: Os blocos verdes são funções a serem enviadas aos núcleos da GPU e os blocos azuis são funções a serem enviadas aos núcleos da CPU.



Fonte – Figura adaptada de (7).

Depois de criar as texturas, independentemente do tipo de simulação (CFD ou matemática) e do tipo de processador (GPU ou CPU), o artista pode realizar mudanças sobre o padrão inicial mediante o uso de mouse e teclado, indicando os parâmetros de configuração tais como:

- Tamanho das gotas de tinta iniciais;
- Cores das tintas;
- Tipos de ferramenta para espalhar as tintas.

Desse modo, os sistemas de marmoreio digital existentes oferecem uma solução próxima ao resultado obtido no processo tradicional, mas nenhum deles fornece uma interface natural apresentando características eficientes, em que o artista crie suas próprias texturas mediante uma interface de fácil aprendizagem, substituindo os dispositivos de entradas comuns (mouse, teclado, o *touchpad* dos *notebooks*, entre outros) pelos movimentos gestuais como por exemplo das mãos, voz, reconhecimento de sons, gestos multi-ponto e reconhecimento facial, entre outros.

1.1 Motivação

A geração de desenhos de marmoreio digital é relevante sobre os seguintes aspectos:

- No processo de marmoreio em papel, a qualidade do resultado obtido é limitada por algumas condições físicas. Por exemplo, o espalhamento que o artista realiza sobre a superfície aquosa depende da disseminação do líquido. Mesmo os fatores ambientais, como a umidade, podem amplamente afetar o resultado final.

- O primeiro sistema de marmoreio digital, *AtelierM* de Mao *et al.* (1), não fornecia uma resposta em tempo real, pois os artistas deviam projetar de antemão todas as operações necessárias, aguardar alguns segundos para obterem o padrão resultante, e em seguida, criarem outro padrão. No entanto, os resultados experimentais no trabalho de Lu *et al.* (9), indicam que o marmoreio automático de imagens pode ser executada em tempo real, mesmo para as imagens com 1680×1200 pixels.

- Existem muitos sistemas de marmoreio digital (detalhados no Capítulo 2) que oferecem alguns padrões de criação, porém o sistema que apresenta maior similitude ao processo de marmoreio de papel (criação, mudanças e obtenção de padrões resultantes) é o proposto por Jin *et al.* (5). Posteriormente, Lu *et al.* (6) detalharam um método de modelagem matemática utilizado na criação dos padrões de marmoreio.

- Os sistemas de marmoreio digital existentes fornecem interfaces em que os dados de entrada são inseridos com o uso de teclado e mouse. Para apresentar uma interface de fácil aprendizagem, segundo Cranor & Garfinkel (10), deve-se aumentar a qualidade de uso do aplicativo que contribui para:

  - Aumentar a produtividade dos usuários;
  - Reduzir o número e a gravidade dos erros cometidos pelo usuário;
  - Reduzir o custo de treinamento;
  - Reduzir o custo de suporte técnico.

Em síntese, os quatro aspectos descritos anteriormente indicam a importância de criar um sistema de marmoreio digital em tempo real, de fácil aprendizagem, considerando as técnicas de Jin *et al.* (5), que permita aos artistas a visualização das mudanças dos padrões gradualmente enquanto realizam várias operações simultaneamente, observando as interações no aplicativo por meio de uma interface em que, se a interação for eficiente, os usuários podem receber apoio computacional para alcançar seus objetivos mais rapidamente, compreendendo melhor as respostas do aplicativo.

1.2 Objetivo

1.2.1 Objetivo geral

O objetivo geral deste projeto é criar uma Interface Natural de Usuário (do inglês, *Natural User Interface* NUI [2]) de fácil aprendizagem, através de uma tela de toques multi-ponto, para o processo de marmoreio digital, fornecendo uma resposta em tempo real dos padrões de marmoreio.

1.2.2 Objetivos específicos

- Adaptar a Dinâmica dos Fluidos Computacional 2D (do Inglês, *Computational Fluid Dynamics* CFD), baseado no *solver* [3] de Stam (12) para criação de tintas digitais seguindo a técnicas de marmoreio digital implementadas por Jin *et al.* (5).
- Melhorar o desempenho na criação de padrões de marmoreio, mediante a geração de um *feedback* visual em tempo real através da simulação feita sobre uma GPU, recebendo apoio computacional para desenhar mais rapidamente os desenhos finais.
- Seguir as diretrizes da Interação Humano-Computador (do inglês, *Human-Computer Interaction* HCI [4]) para realizar a implementação de uma interface mais natural em relação aos sistemas de marmoreio digital existentes.

1.3 Organização do Trabalho

Este trabalho está organizado da seguinte forma:

- **Capítulo 2 - Trabalhos Relacionados**: Apresenta brevemente as principais técnicas envolvidas na simulação do escoamento de fluidos e os sistemas de marmoreio digital estudados para o desenvolvimento deste trabalho.
- **Capítulo 3 - Dinâmica dos Fluidos**: Descreve os aspectos físicos e matemáticos do escoamento de fluidos que serão utilizados na simulação de texturas para nosso protótipo de marmoreio digital.
- **Capítulo 4 - Implementação do Protótipo**: Explica como a interface natural de usuário foi implementada e como foi adaptado o modelo de Stam (12) para simulação das tintas digitais de marmoreio.

[2] É o tipo de interface de usuário em que o usuário interage com um sistema mediante o movimento de gestos (corpo ou mão) (11).

[3] Refere-se a um sistema que resolve um problema matemático através de um programa ou biblioteca computacional, que pode ser aplicada a outros problemas semelhantes.

[4] Disciplina que lida com todos os aspectos do uso de computadores por humanos, geralmente no contexto de sistemas de informação interativos (13).

- **Capítulo 5 - Resultados e Discussões**: Apresenta os padrões de marmoreio digital obtidos por nosso protótipo, mostrando as medições de tempo obtidas nas simulações feitas em laboratório. Além disso, serão apresentadas as discussões dos resultados parciais que foram obtidos no decorrer ao objetivo principal.

- **Capítulo 6 - Conclusão e Trabalhos Futuros**: Explica como o trabalho apresentado permite concluir que cada um dos objetivos foi atingido, deste modo mostraremos quais serão os trabalhos que poderão ser feitos baseados na metodologia desenvolvida na implementação de nosso protótipo.

2 Trabalhos relacionados

2.1 Considerações Iniciais

As metodologias dos trabalhos estudados estão relacionadas segundo como são realizadas as simulações computacionais das tintas de marmoreio em papel. Nesse sentido, as metodologias poderiam se categorizar ou em simulações do escoamento de fluidos ou em simulações baseadas em métodos procedurais. Para uma melhor compreensão destas metodologias e quais trabalhos foram estudados, vamos dividir este capítulo em duas seções: a primeira tem por objetivo apresentar os métodos de simulação utilizados em alguns sistemas de marmoreio digital e a segunda descreve algumas das principais características dos sistemas de marmoreio digital estudados neste trabalho.

2.2 Métodos de Simulação para Marmoreio Digital

Como dito anteriormente, temos dois tipos de metodologias para simulação computacional das tintas de marmoreio em papel, cujas principais características destas metodologias são:

- Simulação do escoamento de fluidos:

  Stam implementou um método simples e eficiente para simulação em tempo real do escoamento de fluidos aplicados em jogos (14). Este método fornece efeitos realistas de redemoinhos de fumaça. A ênfase está na estabilidade e velocidade, o que significa que a sua simulação pode ser confiável com intervalos de tempo arbitrários. A linguagem de programação C é utilizada por Stam para atingir tal simulação e a arquitetura está baseada em grades com tamanhos de 128×128 tanto para 2D como para 3D (12).

  Harris descreve um método para a simulação do escoamento de fluido rápido, estável, que funciona inteiramente na GPU (15), baseado na metodologia de Stam (12). Ele introduz a CFD e a matemática associada, descrevendo em detalhes as técnicas para realizar a simulação na GPU fazendo analogias entre a CPU e a CPU dos elementos usados na simulação.

- Simulação baseada em métodos procedurais:

  A ideia central da metodologia desenvolvida por Lu *et al.* (6) é realizar a simulação computacional das tintas de marmoreio em papel mediante a criação de formas fechadas [1]

[1] É uma expressão matemática que pode ser avaliada em um número finito de operações, ou seja tratável em termos de funções matemáticas elementares.

que serão deformadas enquanto o usuário for inserindo mais tintas, através de um método de modelagem matemática o que é chamado de *desenho gráfico vetorial*. Este método matemático também é usado pelo *software* comercial chamado de *Corel Painter*.

2.3 Sistemas de Marmoreio Digital

Temos classificado os oito sistemas de marmoreio digital estudados segundo as metodologias descritas na seção anterior. A seguir, uma breve descrição de cada sistema.

2.3.1 Baseado na Simulação Física

Estes trabalhos possuem como padrão à simulação Física do escoamento de fluidos, em que, a criação de texturas de marmoreio digital foram feitas com cálculos matemáticos discretizados para a apresentação dos campos de velocidade e densidade.

Até onde sabemos, *AtelierM* (1) foi o primeiro trabalho a propor um sistema de marmoreio digital. Ele foi construído sobre o modelo físico do processo de marmoreio de papel. O processo de criação de padrões marmorizados foi determinado mediante dinâmica dos fluidos computacionais 2D (CFD). No entanto, *AtelierM* não proporciona respostas em tempo real, pois o tempo necessário para cada etapa da criação dos efeitos foi de 32 segundos (feito em um computador Pentium III 933MHz).

Acar & Boulanger (16) apresentaram um modelo de simulação em duas partes. A primeira constitui na solução das equações hidrodinâmicas sobre uma grade relativamente grossa para obter os vetores do fluido. A segunda, implementada em uma grade mais fina, é o modelo de transporte que descreve a evolução da densidade ao longo do espaço das velocidades. A ligação entre estes dois níveis é obtida pela técnica de interpolação de *kriging* (17).

Jin *et al.* (5) implementaram um novo protótipo interativo denominado de *Realtime Marbling* para criação de padrões marmorizados em tempo real, gerados como resultado da advecção das cores em campos de escoamento 2D, obtidos pela resolução dos cálculos numéricos das equações de *Navier-Stokes* na GPU com um *solver multigrid*.

Zhao *et al.* apresentaram o *AtelierM++* (8). Entre as principais contribuições do sistema *AtelierM++* tem-se o *feedback* para o marmoreio digital realizado em tempo real para as texturas de alta resolução e a criação de uma grande quantidade de interfaces intuitivas para um maior controle dos parâmetros de marmoreio, inseridos pelos usuários. Para a simulação, foi usado o modelo físico implementado na linguagem CUDA.

Ando & Tsuruno (18) implementaram uma técnica para criar texturas de marmoreio que podem ser exportadas para um formato de *gráficos vetoriais*, baseada em um método explícito de rastreamento de superfície. Nessa técnica a tinta de marmoreio é representada como um contorno

fechado e advectado pelo escoamento do fluido para formar os redemoinhos, semelhantes aos obtidos no marmoreio em papel.

No ano de 2013, Lu *et al.* (9) apresentaram um novo método de marmoreio em tempo real chamado de *Image Marbleization* que converte uma imagem em uma aparência de marmoreio automaticamente. Sua abordagem pode ser considerada como um método de renderização pictórica da imagem baseada na estilização de marmoreio.

A vantagem dos trabalhos feitos por Jin *et al.* (5) e Lu *et al.* (9) é o uso de programas *shaders* em GPU (Cg) e do trabalho de Zhao *et al.* (8) do uso de algoritmos na GPU a nVidia (CUDA) apresentando um marmoreio digital em tempo real. Os padrões obtidos por alguns destes sistemas de marmoreio digital são apresentados na Figura 4.

2.3.2 Baseado na Simulação Procedural

Nos trabalhos apresentados a seguir o padrão é a simulação baseada em métodos procedurais, que realizam a simulação computacional das tintas de marmoreio em papel como expressões matemáticas para preservar a qualidade ea nitidez do *design*.

O *software* comercial denominado *Corel Painter* (19), oferece duas maneiras de criar um efeito de marmoreio digital usando as opções das paletas do *software*. Um é muito simples e rápido, usando a opção de *Rake Marbling* da paleta *Distortion*. O outro, o que tem mais opções e maiores efeitos, é a ferramenta *Effect Marbling* acessados através do menu *Effects*. Desta forma, os usuários criam suas próprias texturas de marmoreio digital.

Lu *et al.* (6) apresentaram um aplicativo de marmoreio em que não utilizaram a simulação Física dos fluídos, pois as texturas sofrem borramentos nos contornos e os traços finos são perdidos. Portanto, a maior contribuição técnica destes autores é fornecer fórmulas matemáticas para simular o as tintas de marmoreio digital, evitando o custo computacional da simulação Física do escoamento de fluidos.

Os padrões obtidos por estes sistemas de marmoreio digital são mostrados na Figura 5.

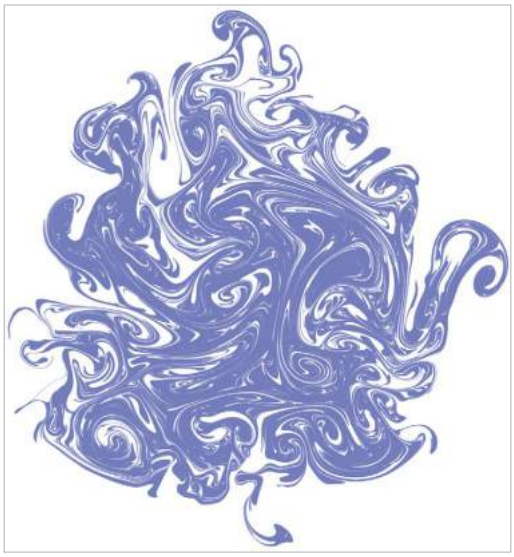
2.4 Considerações Finais

Os trabalhos que foram estudados neste Capítulo fornecem o conhecimento necessário para desenvolver e decidir qual técnica vai ser implementada para o nosso protótipo de marmoreio digital. Deste modo precisamos descrever alguns conceitos envolvidos à Dinâmica dos Fluidos, que é a técnica que acompanha à simulação das tintas digitais para o nosso marmoreio digital.

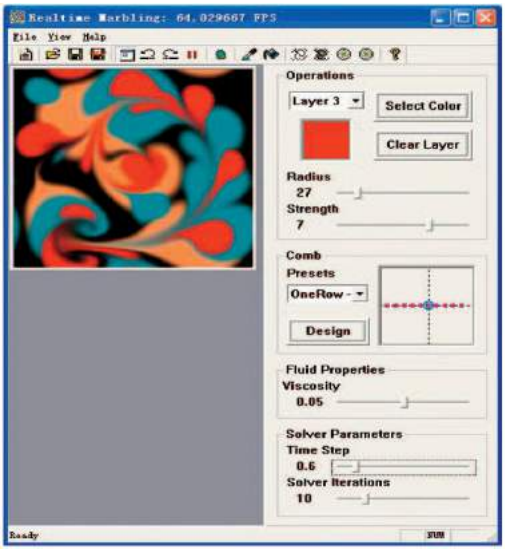
Figura 4 – Sistemas de marmoreio digital baseada na simulação física: (a) Multi-escala, (b) Vetor Gráfico, (c) *Realtime Marbling* e (d) *Image Marbleization*.

(a)

(b)



(c)

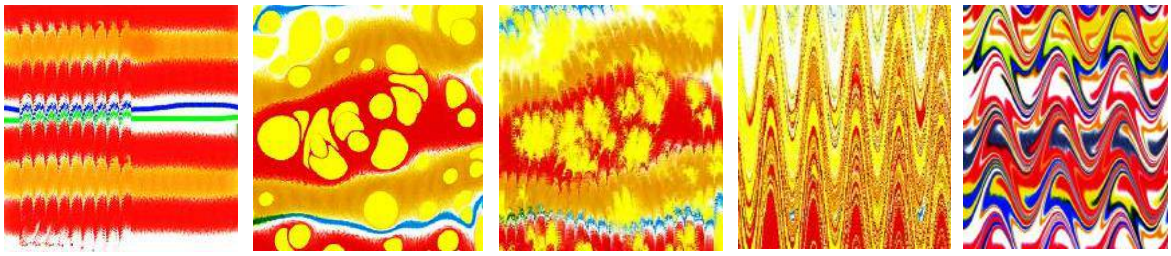


(d)

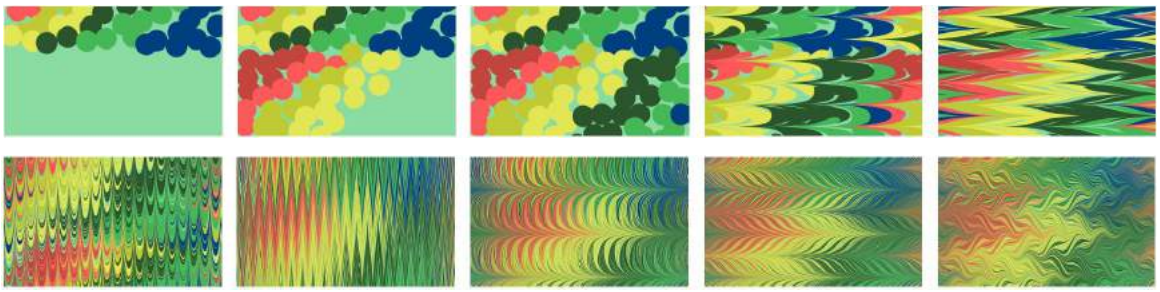
Fonte – (a) (16), (b) (18), (c) (5), (d) (9)

Figura 5 – Sistemas de marmoreio digital baseada em métodos procedurais: (a) *Corel Painter*, (b) Marmoreio Matemático.

(a)



(b)



Fonte – (a) (20), (b) (6)

3 Dinâmica dos Fluidos

3.1 Considerações Iniciais

Como mostrado no capítulo 2, a maioria dos trabalhos relacionados a marmoreio digital estão baseados na simulação Física do escoamento de um fluido para realizar a criação e espalhamento das tintas digitais. Este capítulo tem por objetivo fornecer a base Física e Matemática envolvida no *solver* de Stam para criar e espalhar tintas digitais em nosso protótipo. Portanto, vamos dividir este capítulo em três seções: na primeira introduzimos os principais aspectos físicos que descrevem o escoamento dos fluidos, na segunda descrevemos os componentes da equação de *Navier-Stokes* para que na terceira seção realizemos a sua dicretização.

3.2 Modelagem Física

Os fluidos nos cercam: o ar, os oceanos, a fumaça, o fogo, dentre outros, estão presentes em alguns fenômenos que conhecemos. Para exemplificar os escoamentos de fluidos encontrados na vida cotidiana podemos citar (21):

- Interação de vários objetos com o ar ou água em torno deles;
- Fenômenos meteorológicos (chuva, vento, furacões, inundações, incêndios);
- Processos no corpo humano (o fluxo de sangue, respiração, etc.);
- Aquecimento, ventilação e ar condicionado de edifícios, carros, etc.;
- Combustão em motores de automóveis e outros sistemas de propulsão.

O estudo da dinâmica dos fluidos demanda compreender a evolução temporal e espacial de propriedades de um fluido, como o campo de densidade (ρ), viscosidade (μ), viscosidade cinemática (ν), tensão superficial (σ), compressibilidade, entre outras propriedades que são consideradas ou desprezadas conforme o interesse da simulação. Por exemplo, em nosso trabalho não se faz necessário a simulação da temperatura do fluido.

Vejamos a seguir algumas propriedades mais relevantes para nosso estudo, seguindo as definições de Peixoto & Rodrigues (22).

3.2.1 Densidade

A densidade (ρ) é uma propriedade importante de um fluido. Ela é obtida pelo quociente entre a quantidade de massa m e o volume v que essa quantidade ocupa, assim temos: $\rho = \frac{m}{v}$. Em

geral, usa-se a água como substância de referência gerando assim o conceito *densidade relativa*. Podemos expressar a densidade relativa ρ_r como a razão entre a densidade ρ e a densidade da água, assim temos: $\rho_r = \frac{\rho}{\rho_{agua}}$.

3.2.2 Viscosidade

Uma propriedade envolvendo a física dos fluidos é a *viscosidade*. Um fluido pode ser viscoso ou invíscido (nos fluidos invíscidos não existe arrasto). A viscosidade é uma medida associada à resistência do fluido às deformações. Usaremos a letra grega μ para viscosidade, usualmente com unidades dadas por $[\mu] = \frac{kg}{m \cdot s} = Pas$. Outro conceito relevante é viscosidade cinemática, resultante da razão entre a viscosidade μ e a densidade ρ : $\nu = \frac{\mu}{\rho}$, com unidades em $[\nu] = \frac{m^2}{s}$.

3.2.3 Compressibilidade

A princípio todo fluido é compressível, isto é, alterações de temperatura e pressão podem resultar em alterações na densidade. Porém em alguns casos onde as variações de pressão e temperatura são pequenas pode-se desprezar o efeito de compressibilidade. Neste caso dizemos que o fluido é *incompressível*, isto é, a densidade de uma sub-região fixa do domínio não é alterada durante o escoamento do fluido.

3.3 Modelagem Matemática

As equações que regem a dinâmica dos fluidos são construídas a partir de três princípios físicos fundamentais (23):

1. Conservação da massa;
2. Conservação do momento;
3. Conservação da energia.

Usaremos as equações de *Navier-Stokes*, que asseguram tais princípios, para a simulação do escoamento dos fluidos.

3.3.1 Equações de Navier-Stokes

Na Física é comum fazer suposições simplificando a modelagem de fenômenos complexos (24). Simulação de fluidos não é exceção. Assumimos que o fluído é incompressível e homogêneo.

Um fluido é incompressível (Seção 3.2.3) se o volume de qualquer sub-região do fluido é constante ao longo do tempo. Um fluido é homogêneo se a sua densidade, ρ , é constante no espaço. A combinação de incompressibilidade e homogeneidade significa que a densidade é constante no tempo e no espaço. Esses pressupostos são comuns em dinâmica de fluidos, e eles não diminuem a aplicabilidade da matemática resultante para a simulação de fluidos reais, tais como da água e do ar.

Se a velocidade e pressão são conhecidos para o tempo inicial $t = 0$, então o estado do fluido ao longo do tempo pode ser descrito pelas equações de *Navier-Stokes* para fluxo incompressível (12):

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathrm{F} \tag{1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{2}$$

onde $\mathbf{u}$ é o campo de velocidade, ρ é a densidade (constante) do fluido, ν é a viscosidade cinemática, e $\mathrm{F} = (f_x, f_y)$ representa quaisquer forças externas que atuam sobre o fluido. Observe que a Equação (1) pode ser decomposta em duas equações, considerando que estamos trabalhando no espaço bidimensional. Assim, $\mathbf{u} = (u, v)$ nos garante que:

$$\frac{\partial u}{\partial t} = -(\mathbf{u}.\nabla)u - \frac{1}{\rho}\nabla p + \nu \nabla^2 u + \mathrm{f_x} \tag{3}$$

$$\frac{\partial v}{\partial t} = -(\mathbf{u}.\nabla)v - \frac{1}{\rho}\nabla p + \nu \nabla^2 v + \mathrm{f_y} \tag{4}$$

Assim, temos três incógnitas (u, v, p) e três equações. A seguir, apresentamos os termos da equação de *Navier-Stokes*: advecção, pressão, difusão e forças externas, seguindo o trabalho de Harris (15).

Advecção: A advecção descreve como as quantidades (ou moléculas ou partículas) se movem com o campo de velocidade. Isto quer dizer que se desejamos transportar o valor de densidade (para um ponto fixo), também precisamos transportar todas as outras quantidades associadas com esse ponto. Isso pode incluir atributos físicos, tais como a temperatura e a velocidade, ou atributos especificados pelo problema, como a cor associada àquele ponto. A advecção tem um caso especial e ocorre na transferência do campo de velocidade ao longo dele, o qual é denominado de auto-advecção. O primeiro termo do lado direito da Equação (1) representa essa auto-advecção do campo de velocidade.

Pressão: Também denominada como projeção de pressão. A função da pressão é manter o fluido com o seu volume constante. Quando uma força é aplicada a um fluido, ela não se

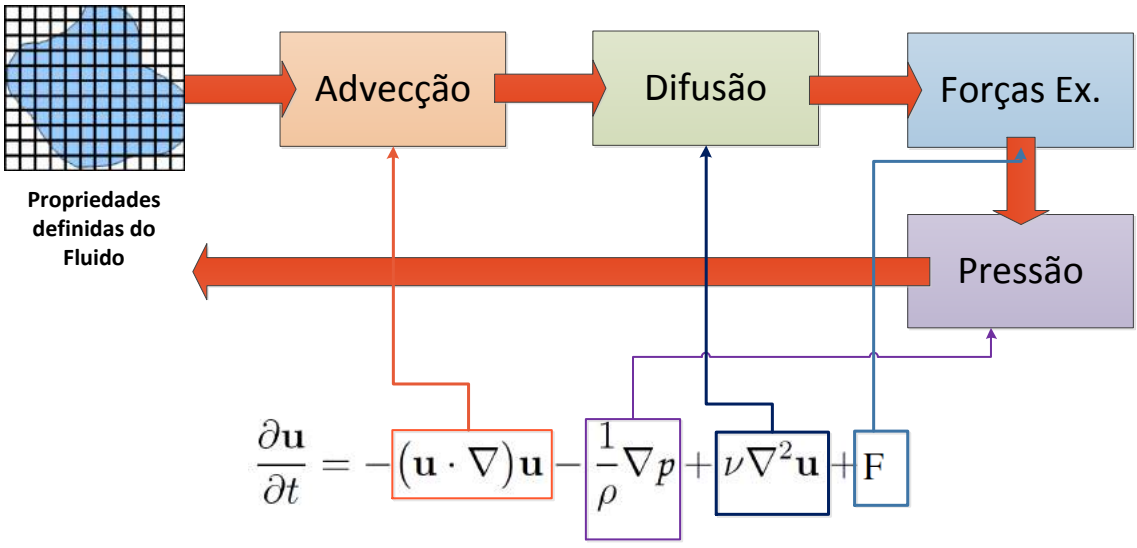
propaga imediatamente através de todo o volume. Em vez disso, as moléculas próximas à força impulsionam sobre aquelas mais distantes, e a pressão se acumula. Devido ao fato da pressão ser a força por unidade de área, qualquer pressão no fluido leva naturalmente a uma aceleração. O segundo termo do lado direito da Equação (1) representa esta aceleração.

Difusão: O terceiro termo do lado direito da Equação (1), a difusão, define como uma quantidade é trocada com os seus vizinhos, em que o movimento das partículas vai de uma região de concentração mais alta para uma de concentração mais baixa. Isto também é referido como o movimento de uma substância a um gradiente de concentração. Um gradiente é a alteração no valor de uma quantidade (por exemplo, pressão, temperatura) com a mudança de uma outra variável (normalmente distância) (24). Por exemplo, uma mudança na pressão ao longo de uma distância é denominada gradiente de pressão, uma mudança na temperatura ao longo de uma distância é denominada gradiente de temperatura.

Forças externas: Existem dois tipos de forças possíveis atuantes sobre o fluido. Por um lado aquelas de longo alcance ($\overrightarrow{F}$) chamadas de forças de volume ou corpo, como a força da gravidade e a força centrifuga, ambas agindo sobre todas as partículas do fluido. Como em geral as variações destes tipos de forças são pequenas no fluido, é comum considerá-las constantes ao longo do mesmo. Por outro lado existem ainda as forças de curto alcance ($\overrightarrow{f}$), ou de superfície. São em geral aquelas geradas devido às tensões, como a força de fricção (22).

A interação entre estes quatro termos é apresentada na Figura (6) e desenvolvida em nosso protótipo no Capítulo 4.

Figura 6 – Estágios da simulação de fluidos - laço principal.



Fonte – Figura adaptada de (25).

3.4 Método Numérico

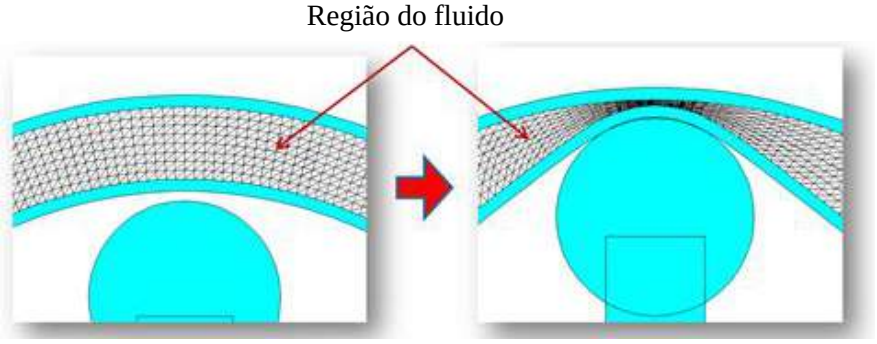
Para simular o escoamento de um fluido é necessário definir primeiramente a discretização do domínio de simulação e dos contornos. Um domínio é discretizado em um conjunto finito de células. O domínio discretizado é chamado de grade ou malha, o qual deve garantir a conservação das equações para massa, momento e energia mediante a discretização em equações algébricas. Todas as equações são resolvidas para apresentar o escoamento do fluido.

O tipo de malha utilizada (retangulares, híbridas, triangulares, etc) está fortemente relacionada aos métodos numéricos utilizados para discretizar as equações que regem a simulação dos fluidos. Segundo os experimentos de Bakker (26) algumas características e tipos de malhas devem ser consideradas:

- Para geometrias simples, malhas de quadriláteros ou de hexaedros [1] fornecem soluções de alta qualidade com menos células do que as malhas simpliciais [2].
- Para geometrias complexas, malhas de quadriláteros ou de hexaedros não apresentam vantagem numérica, e pode-se melhorar os resultados usando malhas simpliciais.
- Um caso especial são as malhas híbridas em que os tipos de malhas podem variar conforme as características locais da geometria.

Assim, por exemplo, temos a representação de uma malha para um fluido na Figura 7-a e na Figura 7-(b) um objeto deformando a malha de um fluido.

Figura 7 – Malha de uma fluido: Representação da interação de um objeto com o fluido.



(a) (b)

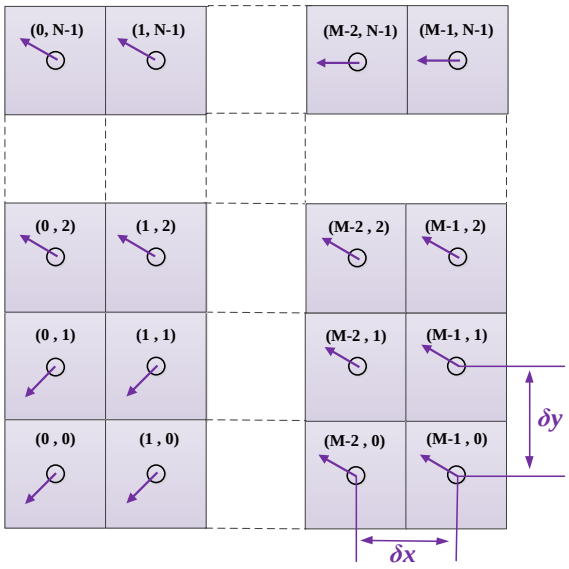
Fonte – (27)

[1] Malha poligonal com faces constituídas de quadriláteros.
[2] Malha poligonal com faces constituídas de triângulos ou de tetraedros.

3.4.1 Discreticação do dominio

Para discretizar o domínio de uma simulação de fluidos e segundo Harris (15), vamos considerar o campo de velocidade (Figura 8), de modo tal que, para cada posição $\mathbf{x} = (x, y)$, há uma velocidade associada no tempo t: $\mathbf{u}(\mathbf{x}, t) = (u(\mathbf{x}, t), v(\mathbf{x}, t))$. Uma vez que temos o campo de velocidade, podemos usá-lo para mover as texturas de densidades exibidas na simulação.

Figura 8 – Grade do campo de velocidade: O estado da simulação do fluido está representada por uma grade $\mathbf{M} \times \mathbf{N}$, em que $\mathbf{M}$ pode ser igual a $\mathbf{N}$. As setas representam a velocidade.



Fonte – Figura adaptada de (15).

3.4.2 Discretização do contorno

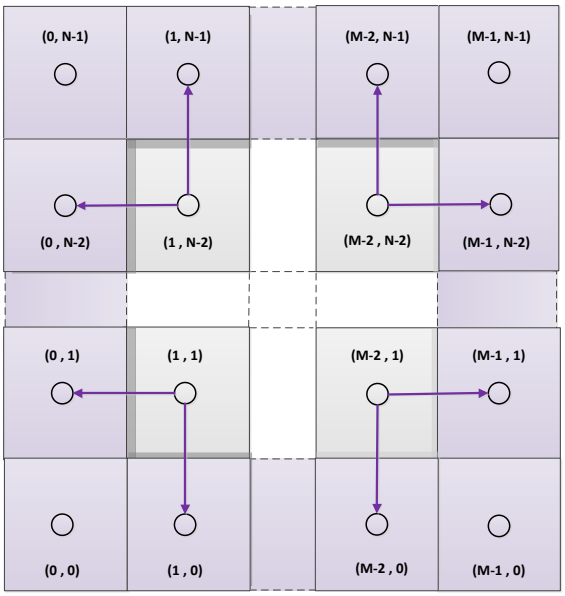
A solução da simulação do escoamento dos fluidos implica definir qual o comportamento do fluido nos contornos. Basicamente, existem três tipos de condições de contornos:

- Condições de tipo *Direchlet*: Com as quais o valor da variável dependente é definida sobre o contorno;
- Condições de tipo *Neumann*: Com as quais o valor das derivadas das variáveis dependentes do contorno é fixo;
- Condições do tipo *Robin* : O também conhecido como do tipo misto, que são combinações dos anteriores.

Para nosso projeto, assume-se que o fluido utilizará as condições de contornos de tipo *Neumann* (Figura 9): nenhum escoamento deve sair dos limites. Isto significa simplesmente

que a componente horizontal da velocidade deve ser zero nas paredes verticais, enquanto que a componente vertical da velocidade deve ser zero nas paredes horizontais. Para densidade e outros campos, deve-se assumir a continuidade (14). Em outras palavras, os fluidos refletem quando elas colidem com o contorno que é o comportamento semelhante ao marmoreio em papel.

Figura 9 – Condições de contorno: Na grade M × N as setas indicam como as células são usadas para copiar valores a partir de dentro dos limites para as células de fronteira.



Fonte – Figura adaptada de (15).

Para o limite de velocidade no lado esquerdo, por exemplo, tem-se:

$$\frac{\mathbf{u}_{0,j} + \mathbf{u}_{1,j}}{2} = 0; \qquad \text{Para } j \in [0, \mathrm{N}] \text{ e N tamanho da grade} \tag{5}$$

3.4.3 Discretização Numérica

As Equações (1) e (2) são utilizadas para representar o estado do fluido ao longo do tempo. Elas contêm três usos diferentes do operador ∇. As três aplicações de *nabla* são: o gradiente, o divergente e o operador Laplaciano. Harris (15) apresenta cada aplicação do operador *nabla* em forma de diferenças finitas (Tabela 1). Desta tabela podemos observar:

- O gradiente de um campo escalar é um vetor de derivadas parciais desse campo escalar;
- O divergente (Equação (2)) é um escalar, calculado, em nosso caso, a partir do produto escalar entre o operador ∇ e o campo de velocidades $\mathbf{u}$;
- Dos itens anteriores, a aplicação do divergente no operador gradiente resulta no operador Laplaciano: $\nabla \cdot \nabla = \nabla^2$.

Tabela 1 – Cálculo dos operadores *nabla* utilizados na simulação de fluidos: Onde i e j referem-se aos dados locais discretos em uma grade cartesiana, e δx e δy são o espaçamento da grade nas dimensões x e y, respectivamente, conforme apresentado na Figura 8.

Operador	**Definição**	**Diferença Finita**
Gradiente	$\nabla p = \left(\frac{\partial p}{\partial x}, \frac{\partial p}{\partial y}\right)$	$\frac{p_{i+1,j} - p_{i-1,j}}{2\delta x}, \frac{p_{i,j+1} - p_{i,j-1}}{2\delta y}$
Divergência	$\nabla \cdot \mathbf{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}$	$\frac{u_{i+1,j} - u_{i-1,j}}{2\delta x} + \frac{v_{i,j+1} - v_{i,j-1}}{2\delta y}$
Laplaciano	$\nabla^2 p = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2}$	$\frac{p_{i+1,j} - 2p_{i,j} + p_{i-1,j}}{\left(\delta x\right)^2} + \frac{p_{i,j+1} - 2p_{i,j} + p_{i,j-1}}{\left(\delta y\right)^2}$

Fonte – (15)

Em nosso protótipo, as células da grade são quadradas ($\delta x = \delta y$), portanto o operador Laplaciano se simplifica a:

$$\nabla^2 p = \frac{p_{i+1,j} + p_{i-1,j} + p_{i,j+1} + p_{i,j-1} - 4p_{i,j}}{\left(\delta x\right)^2} \tag{6}$$

As equações da forma $\nabla^2 f = b$ são conhecidas como Equações de *Poisson* (25). O operador Laplaciano aparecerá tanto na equação de *Poisson* da pressão quanto na equação de *Poisson* da difusão viscosa. Antes de resolver a discretização dessas equações vamos definir a equação da advecção que é o primeiro termo que temos de resolver.

Temos duas opções para calcular a advecção: através do método explícito ou através do método implícito. Com o método explícito a advecção das quantidades são calculadas como uma partícula que se move sobre o passo de tempo atual, porém este método é instável para grandes passos de tempo e velocidades. Por isto, usamos o método implícito que é estável para passos de tempo arbitrários e para maiores velocidades. A advecção é realizada traçando a trajetória da partícula de cada célula da grade ($\mathbf{x}$) para sua última posição (no tempo t) através do campo de velocidade ($\mathbf{u}$), copiamos as quantidades (q) da posição anterior para a célula inicial da grade (no tempo $t + \delta t$). Para este método usamos a seguinte equação:

$$q(\mathbf{x}, t + \delta t) = q(\mathbf{x} - \mathbf{u}(\mathbf{x}, t)\delta t, t) \tag{7}$$

O segundo termo que devemos calcular é uma difusão, na qual os fluidos viscosos têm uma resistência ao fluxo, que resulta em difusão ou dissipação da velocidade. Este termo está relacionada com a equação de *Poisson* para difusão viscosa. Da mesma forma que na advecção, o método explícito é instável para valores grandes da viscosidade cinemática (ν) e do tempo (δt). Portanto deve-se usar o método de diferenças finitas implícito, que é incondicionalmente estável

também utilizado por Stam (12) a fim de conservar a massa:

$$(\mathbf{I} - \nu\delta t\nabla^2)\mathbf{u}(\mathbf{x},t+\delta t) = \mathbf{u}(\mathbf{x},t) \tag{8}$$

onde $\mathbf{I}$ é a matriz identidade, $\mathbf{u}(\mathbf{x},t+\delta t)$ representa o valor que desejamos calcular (difusão viscosa) e $\mathbf{u}(\mathbf{x},t)$ é a advecção no passo de tempo anterior.

O terceiro termo a calcular é a pressão, também conhecida como projeção. A equação de *Poisson* para pressão pode ser obtida mediante o teorema de decomposição de *Helmholtz-Hodge* (15). Este teorema indica, por exemplo, que qualquer campo de vetores (digamos $\mathbf{w}$) pode ser decomposto na soma de dois campos vetoriais: um campo vetorial livre de divergência, e o gradiente de um campo escalar, assim temos:

$$\mathbf{w} = \mathbf{u} + \nabla p \tag{9}$$

onde $\mathbf{w}$ é um campo vetorial do domínio D de nosso fluido, com divergência não-zero, $\mathbf{u}$ tem divergência zero. Se aplicarmos o operador divergente em ambos os lados nesta equação, temos: $\nabla\cdot\mathbf{w} = \nabla\cdot(\mathbf{u}+\nabla p) = \nabla\cdot\mathbf{u} + \nabla^2 p$. Pela Equação (2) em que $\nabla\cdot\mathbf{u} = 0$, finalmente teremos a equação de *Poisson* para pressão (Equação (10)):

$$\nabla^2 p = \nabla\cdot\mathbf{w} \tag{10}$$

As duas equações de *Poisson* ((7) e (10)) podem ser discretizadas usando a Equação 6. O resultado da discretização destas equações é apresentado na Tabela 2. Nesta tabela também esta descrita a equação de Advecção para simular nosso fluido.

Expressamos as equações de *Poisson* (Tabela 2) desta forma porque nos permite usar o mesmo trecho de código para resolver quaisquer das duas equações, executando um número de iterações em cada célula da grade e utilizando os resultados da iteração anterior como entrada para o próximo.

Tabela 2 – Discretização das equações de advecção e *Poisson*.

Operador	Discretização
Advecção	$x_{i,j}^{(k+1)} = \alpha(\gamma x_{i,j}^{(k)} + \theta x_{i+1,j}^{(k)}) + \beta(\gamma x_{i,j+1}^{(k)} + \theta x_{i+1,j+1}^{(k)})$ (11) 1. A partir de uma célula no passo de tempo atual $k+1$, traçamos a posição central da célula para o passo de tempo anterior k através do campo de velocidade; 2. Interpolamos este valor a partir dos dados dos vizinhos; 3. Copiamos a quantidade obtida nessa posição para a célula inicial em estudo; 4. A advecção é feita tanto para a densidade quanto para a velocidade, portanto x representa qualquer desses campos segundo seja o caso, onde $\beta = -u\partial t$, $\alpha = 1-\beta$, $\theta = -v\partial t$ e $\gamma = 1-\theta$ (12).
Eq. *Poisson*	$x_{i,j}^{(k+1)} = \frac{x_{i-1,j}^{(k)} + x_{i+1,j}^{(k)} + x_{i,j-1}^{(k)} + x_{i,j+1}^{(k)} + \alpha b_{i,j}}{\beta}$ (12) 1. As duas equações de *Poisson* serão resolvidas mediante o método das iterações *Gauss-Seidel* (12); 2. α e β são constantes. Para cada equação os valores de x, b, α e β são diferentes; 3. Para *Poisson-pressão*: x representa p, b representa $\nabla \cdot w$, $\alpha = -(\delta x)^2$, e $\beta = 4$; 4. Para difusão viscosa: tanto x quanto b representam $\mathbf{u}$, $\alpha = \frac{(\partial x)^2}{v\partial t}$, $\beta = 4+\alpha$.

Fonte – Tabela adaptada de (15).

3.5 Considerações Finais

O propósito deste capítulo foi apresentar a ferramenta principal de nosso projeto, a simulação do escoamento dos fluidos, o qual foi descrito mediante duas modelagens: Física e Matemática. Na primeira definimos as propriedades do escoamento dos fluidos e na segunda as equações, e as suas respectivas discretizações, que simulam essas propriedades.

As equações de *Navier-Stokes* fornecem os principais fatores (incompressibilidade, advecção, difusão, pressão e forças), elas nos ajudam a simular o escoamento do fluido envolvido em nosso projeto para a simulação das tintas em nosso marmoreio digital. Para implementar estas tintas em nosso protótipo, devemos utilizar as discretizações apresentadas na Seção 3.3 que descreve os métodos numéricos da resolução das principais equações.

Com estes estudos realizados além dos apresentados no Capítulos 2, podemos explicar nossa solução proposta, a mais adequada para simulação de tintas de marmoreio digital. Esta proposta implica a resolução em tempo real das equações de *Navier-Stokes* utilizadas na simulação do escoamento de fluidos por Stam.

4 Implementação do Protótipo

4.1 Condições Iniciais

Para o desenvolvimento de nosso protótipo decidimos que é adequado dividi-lo em cinco etapas, as quais foram definidas seguindo as técnicas de marmoreio digital implementadas por Jin *et al.* (5) e Lu *et al.* (6):

1. Implementar um sistema interativo de fácil aprendizado para os usuários;
2. Simular e paralelizar (de CPU para GPU) o escoamento de texturas baseado no *solver* de Stam;
3. Realizar a advecção das cores das tintas digitais, conservando que elas não se misturem;
4. Corrigir os borramentos ocorridos nos contornos das tintas digitais;
5. Implementar os padrões para aplicar os espalhamentos sobre as tintas digitais.

4.2 Sistema interativo

Nesta primeira etapa de desenvolvimento vamos descrever o protótipo através das classes e métodos principais, a relação entre eles mediante diagramas e discutiremos sobre a biblioteca multi-pontos de Qt utilizada para a interação com os usuários. Previamente a isto vamos indicar quais ferramentas foram usadas na implementação do sistema computacional interativo.

4.2.1 Ferramentas

As ferramentas utilizadas para o desenvolvimento de nosso protótipo são:

Qt *Framework* : Este *framework* vai proporcionar o *designer* para criar as interfaces necessárias na interação com os usuários, pois ela permite um desenvolvimento fácil em interfaces multi-plataforma utilizando a linguagem de programação C++. Além da interface gráfica, o Qt vai fornecer o módulo de reconhecimento de toques multi-pontos através da classe `QTouchEvent`. Este *framework* vem integrado com OpenGL, além disso tem a capacidade de executar arquivos na linguagem GLSL para comunicação com a GPU.

Qt5 Visual Studio Add-in: Esta ferramenta permite criar, depurar e executar aplicativos Qt a partir de versões que não sejam *Express* do Microsoft Visual Studio 2008, 2010 e 2012. O add-in contém assistentes às ferramentas gráficas do projeto Qt e o gerenciador de recursos Qt integrado e automatizado.

Biblioteca Cinder: É uma biblioteca multiplataforma da linguagem de programação C++, a fim de programar com intenção estética; este tipo de desenvolvimento muitas vezes é chamado de codificação criativa. Isso inclui ambientes tais como gráficos, áudio, vídeo e geometria computacional. Ela é compatível com as bibliotecas Qt e OpenGL, usadas em nosso protótipo.

Microsoft Visual Studio: É o compilador da linguagem de programação C++, a fim de usá-lo na integração das biblioteca Cinder e o Qt *Framework*. É importante para uma boa integração destas ferramentas realizar as configurações adequadas.

Tabela 3 – Bibliotecas e Ferramentas do Protótipo com suas respectivas versões.

Ferramenta/Biblioteca	**Versão**
Qt Framework	5.4.2 - x86
Qt5 Visual Studio Add-in	1.2.5
Cinder	0.8.5
Microsoft Visual Studio	2010 Professional

4.2.2 Descrição de Classes e Métodos

MainWindow: Através dos objetos instanciados a esta classe conseguimos gerenciar nosso protótipo. Mediante esta classe, a interface gráfica e as classes principais de nosso protótipo são relacionados. Como a nossa interface apenas apresenta botões pelos quais são realizados todas as comunicações com as classes restantes, esta classe não apresenta métodos implementados, apenas são usados os *Signals* [1] e *Slots* [2].

Solver: Através dos objetos instanciados a esta classe realizamos os cálculos necessários para resolver em tempo real as equações de *Navier-Stokes*, ou seja a responsável pela simulação do escoamento de fluido tal como foi descrito no Capítulo 3. Para realizar esta simulação, a classe utiliza *frame buffer objects* (FBO) para realizar o *rendering* [3] capturando os resultados parciais (texturas) em cada passo de tempo.

`iterateModel()`: Permite atualizar os componentes do fluido, ou seja da velocidade e da densidade, através dos métodos `calculateNextVelocity()`, `calculateNextDensity()`. Este método é invocado pela classe `OpenGLWidget` para atualizar o *widget* usado como superfície de *design*.

[1] É usado para comunicação entre objetos Qt, além disso é emitido quando ocorre um evento particular.
[2] É usado para comunicação entre objetos Qt, além disso é uma função que é chamada em resposta a um determinado *Signal*.
[3] É o processo de gerar uma imagem a partir de um modelo 2D ou 3D (chamado de cena) por meio de programas de computador.

calculateNextVelocity(): Gerencia os passos necessários para gerar a textura da velocidade. Os resultados de cada passo são armazenados temporalmente num FBO chamado de `tempBuffer`.

calculateNextDensity(): Gerencia os passos necessários para gerar a textura da densidade. Os resultados de cada passo são armazenados temporalmente num FBO chamado de `tempBuffer`.

addVelocity(): Permite adicionar as forças externas sobre a tela. Estas forças se tornariam nos espalhamentos sobre as tintas digitais criadas na tela multi-ponto de nosso protótipo, por isto é necessário que este método seja invocado pela classe `OpenGLWidget` para a interação com os usuários mediante os toques sobre a tela.

addDensity(): Assim como o método `addVelocity()`, o método `addDensity()` adiciona forças externas em nossa simulação, porém ela cria mediante estas forças as tintas digitais. Este método é invocado em razão à quantidade de toques e as cores selecionadas pelo usuário, por isto é necessário método seja invocado pela classe `OpenGLWidget` para a interação com os usuários mediante os toques sobre a tela.

getTextureDensity(): Permite obter a textura da densidade mediante o número da camada [4] (cada camada representa uma cor distinta) à qual pertence a densidade solicitada.

Gradiente: Através dos objetos instanciados a esta classe gerenciamos as cores que serão utilizadas para realizar a criação das tintas digitais. Esta classe está baseada no *sketch* implementado por Phagor (28) para simulação do escoamento de fluidos coloridos. A importância de utilizar gradientes de cores é dar maior realismo às tintas digitais criadas pelo usuário. Estas cores são geradas e armazenadas na CPU (memória RAM).

makeDefaultGradient(): A tarefa deste método é normalizar as cores selecionadas pelos usuários, a fim reservar os nós em que serão armazenadas as cores. Normalizar implica na conversão dos componentes da cor da classe `QColor` do Qt ao tipo `float` da linguagem de programação C++.

addNode(): É invocado pelo método `makeDefaultGradient()` a fim de armazenar os nós das cores normalizadas, inserindo estes nós na memória RAM.

getColor(): Permite realizar os cálculos necessários para obter as transições das cores que compõem o gradiente (previamente armazenadas em nós). Todo gradiente de nosso protótipo está composto por transições de tons de uma cor.

fillVectorOfColors(): Obtém as cores geradas pelo método `getColor()`, a fim de inserir estas cores num vetor e assim obter a cor selecionada pelos usuários em forma de gradiente.

[4] Este conceito será detalhado na Seção 4.4

OpenGLWidget: Através dos objetos instanciados a esta classe recebemos os *signals* da classe `MainWindow` através dos *slots*, a fim de realizar o *feedback* aos usuários, mediante a classe `QOpenGLWidget` do Qt.

`initializeGL()`: Este é um método virtual da classe `QOpenGLWidget` do Qt e é chamado uma vez, antes da primeira chamada dos métodos `paintGL()` ou `resizeGL()`. Este método deve configurar qualquer recurso necessário para utilizar a biblioteca gráfica de OpenGL.

`resizeGL()`: Este é um método virtual da classe `QOpenGLWidget` do Qt e é chamado sempre que o *widget* foi redimensionado. Tem dois parâmetros do tipo `int`, a fim de serem passados como o novo tamanho.

`paintGL()`: Este é um método virtual da classe `QOpenGLWidget` do Qt e é chamado sempre que o *widget* precisa ser pintado (atualizar desenho na tela).

`event()`: Este é um método virtual de tipo booleano da classe `QObject` que recebe eventos para um objeto (para nosso caso é um *widget*) e deve retornar `true` se o evento e foi reconhecido e processado.

`CreateVBOs()`: Permite criar os objetos *buffer* (VBO) para gerenciar os atributos dos vértices na GPU, ou seja a manipulação sobre a posição, cores, índices, coordenadas de texturas, etc.

`CreateShaders()`: Permite criar em tempo de execução os programas *shader*. O Cinder fornece a classe `GlslProg` para a criação destes programas e para compilá-los através da linguagem GLSL, mediante o *vertex shader* ou *fragment shader*.

`drawFluidLayer()`: Permite desenhar as tintas digitais no *widget*. Este método é chamado pelo método `paintGL()`, a fim de pintar o *widget* em cada modificação feita pelo usuário, por meio da obtenção das texturas das densidades geradas pela classe `Solver`.

Os processos do nosso sistema de marmoreio efetuam comunicações e intercâmbios de dados entre a interface do usuário e a simulação do escoamento do fluido. Cada processo esperado leva a uma sequência distinta de tarefas. A Figura 10 – através de um diagrama de colaboração – descreve um primeiro processo, que é a inserção da primeira gota de tinta. Concisamente, as tarefas são:

Preparando o sistema para receber uma gota de tinta: O processo começa com o usuário seleciona a ferramenta conta-gotas do protótipo (Item 1). Esta informação dispara todos os sub-processo para simular a gota de tinta no solver do escoamento do fluido, bem como para renderizá-lo (Itens 2 e 3).

Simulação do Fluido: Neste momento, começa a simulação do escoamento do fluido. O item 4 refere-se à parte do solucionador que chama um loop (Itens 5 e 6) para calcular as Equações de *Navier-Stokes*.

Transformação de Domínio e Cálculo da Densidade: O *solver* do escoamento de fluido está

preparado para receber as entradas do usuário, que são transformadas nas coordenadas de simulação de domínio (Item 7). Neste caso, o sistema está preparado para receber a gota de tinta, o que significa calcular os valores da densidade e armazená-los em um *frame buffer object* (29) (Item 8).

Configuração de Objetos Gráficos: Os objetos de textura e os *shaders* (29) são criados (Itens 9 e 10) para renderizar os fluidos.

De Fluido à Shader de Textura: Para renderizar a gota de tinta, nossa classe `OpenGL` requer o *frame buffer object* que armazena a densidade do *solver* do escoamento do fluido (Item 11). O *solver* então envia o *buffer* requerido como uma textura (Item 12). Observe que a classe `OpenGL` deve especificar a camada que ele renderizará (paintID, Item 11).

Gradiente de Color: Em paralelo, para melhorar a qualidade da renderização, é criada um *lookup table* para simular ondulações fluidas e interações de luz (Itens 13-15). Este *lookup table* é enviado para a GPU como uma textura (Item 16).

Resultados: Esta última etapa é responsável por combinar na GPU tanto a textura da densidade quanto a textura do gradiente para exibir a cor da tinta (Item 17).

Figura 10 – Diagrama de Colaboração: Criar tinta da cor azul.

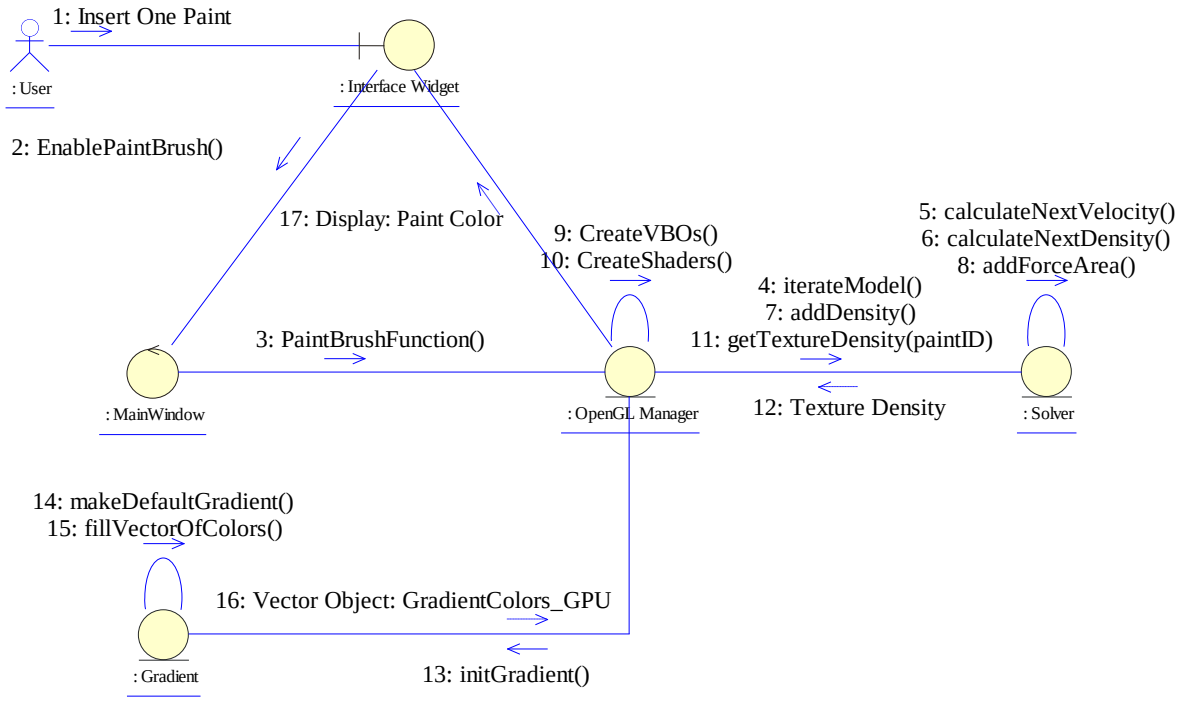
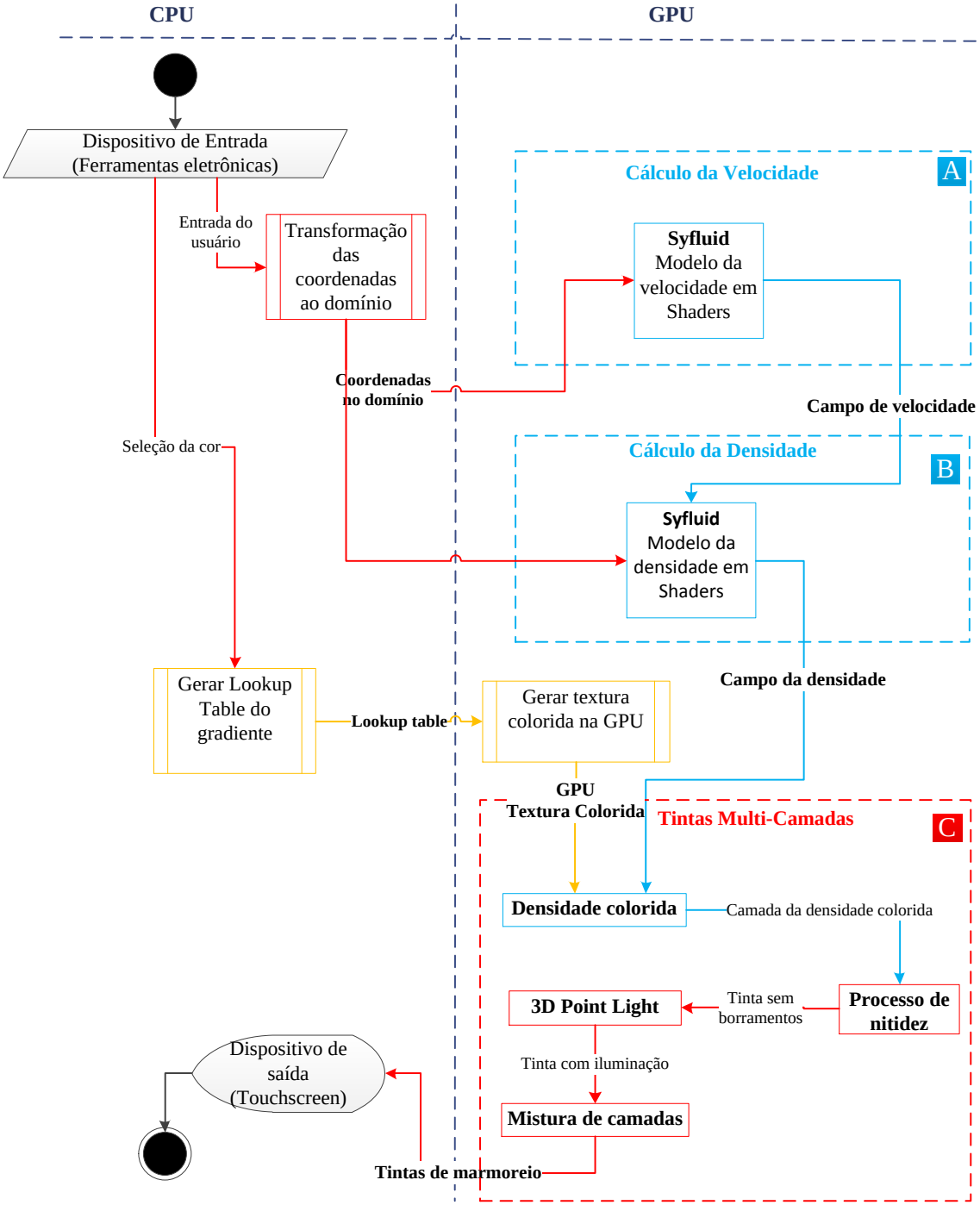


Figura 11 – Arquitetura de nosso marmoreio digital: Distribuição de tarefas de cada processador, em que a interação entre *shaders* é realizada no processador GPU.



4.2.3 Arquitetura

Uma vez que a simulação de fluxo de fluido é executada na GPU, há uma interação entre todos os programas *shaders* (tanto para renderização quanto para simulação de fluidos) usados para fornecer um *feedback* em tempo real ao usuário. Todo o processo é sistematizado na Figura

11. Os objetos azuis claros representam rotinas da biblioteca de fluxo de fluidos (*syfluid* (30)) que exploramos e adaptamos ao nosso sistema, os objetos amarelos são responsáveis por criar as tintas com ondulações e interações de luz. Para esses propósitos, empregamos e adaptamos as rotinas fornecidas em (28). Os objetos vermelhos correspondem às nossas abordagens de comunicação e renderização.

A CPU é responsável principalmente por duas tarefas: as transformações do domínio a partir da posição do usuário nos dispositivos de entrada eletrônica para a simulação do domínio de fluxo de fluido e a geração das tabelas de pesquisa de gradiente usadas para simular ondulações de pintura e iluminação. O processador GPU é responsável pelo processo de três núcleos, cada um representado por um bloco na Figura 11.

O Bloco A visa criar a textura da velocidade e assegurar a propriedade da incompressibilidade. Neste bloco, são recebidos os vetores de força de deslocamento fornecidos pela interação dos usuários na interface. Este bloco é o primeiro a ser executado e o campo de velocidade resultante deste processo é enviado para o Bloco B.

O Bloco B tem como objetivo criar a textura de densidade, ou seja, as tintas digitais que serão geradas de acordo com as cores selecionadas pelos usuários. Esse processo é o segundo a ser executado. Ambos os blocos A e B têm rotinas que empregam programas *shaders* que resolvem as Equações (1) - (2) descritas no Capítulo 3. Cabe ressaltar que os *shaders* desta biblioteca foram adaptados ao nosso propósito de suportar densidades multi-camadas, permitindo manipular efetivamente várias tintas coloridas simultaneamente.

O Bloco C fornece a renderização de tinta multi-camada (Sec. 4.4). Este bloco processa duas texturas de entrada: uma para a cor atribuída à tinta (representada mediante a seta **GPU Textura Colorida** da Figura 11) e outra para a densidade de dados de fluidos obtida a partir do Bloco B (representada mediante a seta **Campo da densidade** da Figura 11). Essas entradas são adequadamente misturadas para definir a tinta de marmoreio final.

4.3 Simulação das Texturas

Antes de prosseguir, devemos definir o que é uma textura. Como visto no Capítulo 3, o domínio da simulação será discretizado em uma grade bidimensional. A representação para esta grade na CPU é uma matriz. O análogo a uma matriz na GPU é uma textura. Para ambos os casos, são utilizadas para armazenarem a densidade e velocidade do fluido. Portanto, usaremos o termo textura de densidade para definir as tintas do marmoreio digital e a textura da velocidade para definir os espalhamentos das tintas, chamados de padrões de marmoreio.

Quanto ao domínio do fluido, a representação pode ser feita principalmente por dois métodos: o método Lagrangiano (24) ou o método Euleriano (24). O primeiro faz a simulação sobre um sistema de partículas, em que, cada ponto é tratado como uma partícula separadamente,

as quais se movem e interagem entre si. No segundo, também conhecido como método de grades, o espaço do fluido é dividido em células individuais e cada uma delas contêm todas as informações do fluido. Nossa simulação será feita como no método Euleriano, mas o cálculo das equações serão feitas como no método Lagrangiano. Esta técnica é chamada de método Semi-Lagrangiano (24).

As texturas que serão simuladas em nosso aplicativo de marmoreio digital serão: densidade e velocidade. Elas serão apresentadas mediantes listagens que seguem a estrutura da linguagem C++ e para realizá-lo em tempo real vamos a utilizar a linguagem GLSL. Cabe ressaltar que as texturas de densidade a simular nesta seção são texturas em branco e preto, portanto seguindo o trabalho de Jin *et al.* (5) vamos introduzir o conceito de *Multi-camada* na Seção 4.4.2, porém que será utilizada nesta seção a fim de indicar que uma camada representa uma cor independente para as tintas digitais.

Outra consideração importante é que o processamento das texturas será executado na GPU por meio de programas *shaders* usando tanto o *vertex shader* quanto o *fragment shader*. Como a simulação de nossa solução na CPU executa os passos do algoritmo em *loops*, usando um par de *loops* aninhados para iterar cada célula da grade, na GPU usaremos o *fragment shader*. Este não tem a capacidade de executar o *loop* interno sobre cada *texel*[5] em uma textura. No entanto, o *fragment shader* foi concebido para realizar cálculos idênticos em cada fragmento. Nós assumiremos como se houvesse um processador para cada fragmento, e que todos os fragmentos são atualizados simultaneamente.

O *vertex shader*, apresentado na Listagem 1, tem três variáveis de entrada (Linhas 1, 2, e 3) que representam a matriz de transformação, os vértices e as coordenadas de textura. Esta última variável é interpolada para o *fragment shader*, contendo a posição do vértice atual (Linha 9).

Listagem 1 – Vertex Shader: vshader.glsl.

```
1  uniform mat4 modelView;
2  in vec4 vPosition;
3  in vec2 vcoordText;
4
5  out vec2 fcoordText;
6
7  void main ()
8  {
9      fcoordText = vcoordText;
10     gl_Position = modelView*vPosition;
11 }
```

[5] Unidade mínima de uma textura.

4.3.1 Textura de Densidade

Vamos simular a textura de densidade movendo-se através da textura de velocidade, no qual a densidade:

- Aumenta devido às fontes inseridas pelo usuário (a chamar de adição de fonte);
- Difunde-se a uma determinada taxa (a chamar de difusão);
- Segue o campo de velocidade (a chamar de advecção).

Portanto, o processo de criação de texturas de densidade descrito na Listagem 2, resolve repetidamente as três etapas descritas acima para cada passo de tempo, obtendo uma textura final (`density`) gerada a partir de uma textura inicial (`density_prev`). Na Linha 3 o *loop* `for` controla a quantidade das camadas de texturas de densidade que serão criadas mediante a variável `numColorLayer`. Nas Linhas 5-7 temos cada uma das etapas da criação de texturas de densidade, agrupadas numa rotina. Na Linha 5 a variável `Diff` representa a taxa com que as texturas serão espalhadas. Como na última etapa é realizado o transporte das texturas de densidade, precisamos das texturas de velocidade (`u` e `v`) para atingir esta etapa (Linha 7). Cabe ressaltar que tanto as densidades (`density`, `density_prev`) quanto as velocidades (`u`, `v`) são *buffers* FBO do tipo `gl::Fbo` da biblioteca Cinder.

Listagem 2 – Solver.cpp: método `calculateNextDensity()`.

```
1 void Solver::calculateNextDensity()
2 {
3   for(unsigned int i=0; i<numColorLayer; i++)
4   {
5     addSource(density[i], density_prev[i]);
6     diffuse(density_prev[i], density[i], 0, Diff);
7     advect(density[i], density_prev[i], 0, u, v);
8   }
9 }
```

Fonte – Listagem adaptada de (30).

Adição de fonte: As fontes são inseridas pelo usuário mediante os toques sobre nossa tela multi-ponto. Este método gerenciado pela classe `Solver.cpp` é apresentado na Listagem 3, onde armazenamos a adição das novas texturas de densidade. Na Linha 5 temos o FBO chamado de `tempBuffer` que vai copiar temporalmente os *texels* da densidade final `tex` mediante a função `blitTo()` da biblioteca Cinder, para ser usado como auxiliar na adição *texel* por *texel* da densidade inicial `tex0` em cada passo de tempo `dt`, fazendo uso do programa *shader* `addSourceShader` que é um objeto do tipo `GlslProg` da biblioteca Cinder (Linhas 8-15). O *rendering* do resultado obtido é feito mediante o método `render()` (Linha 18).

Listagem 3 – Solver.cpp: método `addSource()`.

```
1  void Solver::addSource(GPUBuffer &tex, GPUBuffer &tex0)
2  {
3    //copy density to tempBuffer: copy a block of pixels from
4    //the read framebuffer to the draw framebuffer
5    tex.blitTo(tempBuffer,tex.getBounds(),tempBuffer.getBounds());
6
7    //Bind shader and pass variables as uniforms
8    addSourceShader.bind();
9    addSourceShader.uniform("tex", 0);
10   addSourceShader.uniform("tex0", 1);
11   addSourceShader.uniform("dt", delta);
12
13   //Bind textures
14   tempBuffer.getTexture().bind(0);
15   tex0.getTexture().bind(1);
16
17   //rendering
18   render(tex);
19
20   //clean all
21   tempBuffer.getTexture().unbind();
22   tex0.getTexture().unbind();
23   addSourceShader.unbind();
24 }
```

Fonte – Listagem adaptada de (30).

O arquivo *shader* de adição de fontes executado na GPU é apresentado na Listagem 4. Nele as texturas são representadas mediante o tipos de variável `sampler2D` tanto pra densidade inicial (`tex0`, Linha 1) quanto pra densidade final (`tex`, Linha 2). As coordenadas da textura foram interpoladas no *vertex shader* (Listagem 1) e enviadas ao *fragment shader* (Linha 4). Mediante a função `texture` (Linhas 9 e 10) recuperamos os *texels* das texturas de densidade e aplicamos o passo do tempo `dt` fim de obtermos a nova textura de densidade a partir dos valores iniciais. Finalmente as cores obtidas das densidades são desenhadas (armazenadas) no fragmento (Linha 12).

Listagem 4 – *Fragment Shader* de Adição de Fontes: faddsource.glsl.

```
1  uniform sampler2D tex0;
2  uniform sampler2D tex;
3  uniform float dt;
4  in vec2 fcoordText;
5  out vec4 outColor;
6
7  void main()
8  {
9    vec4 DTexel = texture(tex, fcoordText.xy);
10   vec4 DPTexel = texture(tex0, fcoordText.xy);
11   DTexel += dt * DPTexel;
12   outColor = DTexel;
13 }
```

Fonte – Listagem adaptada de (30).

Difusão: A segunda etapa, apresentada na Listagem 5, corresponde à difusão da textura de densidade a uma determinada taxa de difusão, em que esta textura vai se espalhar através dos vizinhos mais próximos da célula (*texel*) em estudo. As texturas que serão utilizadas são `tex0` e `tex` que representam à densidade inicial e final respectivamente. A seguir vamos descrever este processo desenvolvido na Listagem 5.

O processo de difusão começa quando finaliza o processo de adição das fontes. O resultado desta última etapa foi renderizado no FBO da densidade (`density`) para ser utilizado na difusão, em que o método `diffuse()` realiza esta tarefa. Os parâmetros são enviados pelo método `calculateNextDensity()` em que `density` e `density_prev` são enviados como parâmetros de referência a `tex0` e `tex` respetivamente, `b` é inicializado com 0 para diferenciar que a tarefa de difusão será realizada com a densidade e não com a velocidade. Além disso a taxa de difusão `actual_diffuse` é gerenciada por este método.

Como dito no capítulo 3 a difusão foi discretizada na Equação 12 para resolver um sistema linear com densidade (`tex`) desconhecida; esta tarefa é feita pelo método `linsolve()` (Linhas 7-38). É necessário realizar o cálculo das constantes α e β da Equação 12, para isto, na Linha 3-4 calculamos estas constantes em que α = `a` e β = `1+4*a`.

O processo iterativo começa na Linha 9, com o *loop* `for` de 20 iterações que é o valor em que atingimos a convergência. O processo de difusão assim como o processo de adição de fontes utilizam programas *shaders*; na difusão este programa é o `linSolveShader` e realiza a cópia temporal da densidade inicial `text0` ao FBO `tempBuffer` (Linha 12) para usá-lo como auxiliar nos cálculos realizados na GPU (Linhas 15-25), fazendo uso da densidade final `tex` para finalmente realizar o *rendering* dos resultados obtidos, ou seja é apenas no *rendering* que fazemos a projeção dos resultados obtidos na resolução do sistema linear. Por último, na Linha 36, gerenciamos os contornos da região da simulação em que é usado o método `setBnd()`, o qual será detalhado na Seção 4.3.3.

Listagem 5 – Solver.cpp: métodos `diffuse()` e `linSolve()`.

```
1  void Solver::diffuse(GPUBuffer &tex0, GPUBuffer &tex, int b, float
       actual_diffuse)
2  {
3    float a = dt * actual_diffuse * N * N;
4    linSolve(tex0, tex, b, a, 1 + 4 * a);
5  }
6
7  void Solver::linSolve(GPUBuffer &tex0, GPUBuffer &tex, int b, float a,
       float c)
8  {
9    for (int k = 0; k < 20; k++){
10     //copy tex0 to tempBuffer: copy a block of pixels from
11     //the read framebuffer to the draw framebuffer
12     tex0.blitTo(tempBuffer,tex0.getBounds(),tempBuffer.getBounds());
13
14     //Bind shader and pass variables as uniforms
15     linSolveShader.bind();
16     linSolveShader.uniform("tex0", 0);
```

```
17      linSolveShader.uniform("tex", 1);
18      linSolveShader.uniform("a", a);
19      linSolveShader.uniform("c", c);
20      linSolveShader.uniform("OneNStep", OneNStep);
21      linSolveShader.uniform("OnePrevOneNStep", 1.0f - OneNStep);
22
23      //Bind textures
24      tempBuffer.getTexture().bind(0);
25      tex.getTexture().bind(1);
26
27      //rendering: for all cell from 1..N:
28      render(tex0);
29
30      //clean all
31      tempBuffer.getTexture().unbind();
32      tex.getTexture().unbind();
33      linSolveShader.unbind();
34
35      //manage boundaries
36      setBnd(tex0, b);
37      }
38  }
```

Fonte – Listagem adaptada de (30).

O arquivo *shader* de difusão executado na GPU é apresentado na Listagem 6. Nele as texturas são representadas mediante o tipos de variável `sampler2D` tanto pra densidade inicial (`tex0`, Linha 1) quanto pra densidade final (`tex`, Linha 2). As constantes `OneNStep` e `OnePrevNStep` são usadas como delimitadores a fim de que os vizinhos de uma célula em estudo não sejam *texels* não existentes (controlado mediante o condicional `if` na Linha 11). Como observado na Linha 14 temos descrita a Equação 12 usando a função `texture` para analisar cada *texel* da textura inicial; este resultado é enviado ao fragmento mediante a variável `outColor`. Finalmente, caso seja um *texel* que pertence ao contorno, não é realizada a resolução da equação (Linha 16).

Listagem 6 – *Fragment Shader* de Difusão de Texturas: flinsolve.glsl.

```
1  uniform sampler2D tex0;
2  uniform sampler2D tex;
3  uniform float a;
4  uniform float c;
5  uniform float OneNStep;
6  uniform float OnePrevOneStep;
7  in vec2 fcoordText;
8  out vec4 outColor;
9
10 void main(){
11   if (all(greaterThan(fcoordText.xy, vec2(OneNStep, OneNStep))) &&
12     all(lessThan(fcoordText.xy, vec2(OnePrevOneStep, OnePrevOneStep))))
13
14     outColor = (texture(tex,fcoordText.xy) + a * (texture(tex0,fcoordText.
           xy + vec2(-OneNStep, 0.0)) + texture(tex0,fcoordText.xy + vec2(
           OneNStep, 0.0)) + texture(tex0,fcoordText.xy + vec2(0.0, -OneNStep))
            + texture(tex0,fcoordText.xy + vec2(0.0,  OneNStep)))) / c;
15   else
16     outColor = texture(tex0, fcoordText.xy);
17 }
```

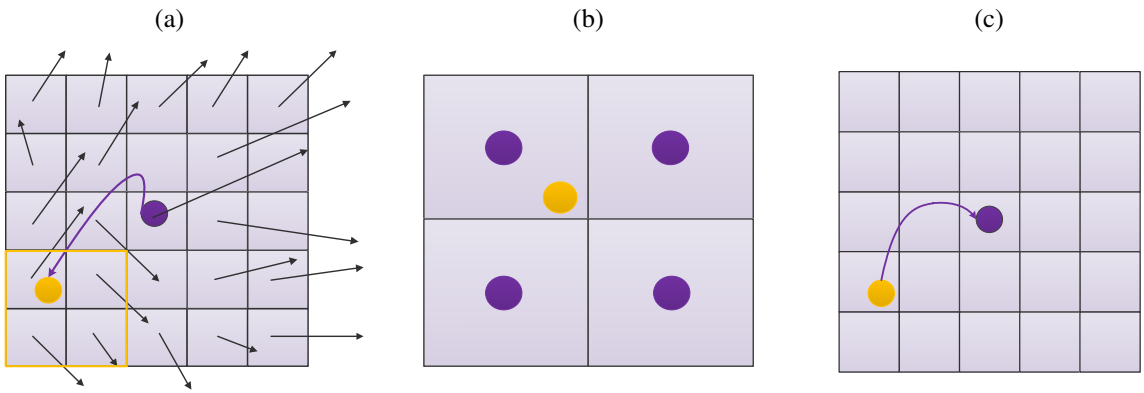
Fonte – Listagem adaptada de (30).

Advecção: Na terceira etapa, Stam (12) apresenta um método de resolução da advecção da textura de densidade por meio da textura de velocidade que é representada tanto no eixo *x* quanto no eixo *y* através dos FBO `u` e `v` respectivamente, para isto apresentamos este método na Listagem 7. Este processo é realizado em três passos (Figura 12), que consistem em: traçar o *texel* para trás através da velocidade; interpolar o valor atual a partir do valor anterior; atribuir este valor ao *texel* atual. A seguir vamos a descrever o processo da advecção seguindo a Listagem 7.

O processo da advecção começa com o resultado obtido pelo processo da difusão que foi renderizado no FBO `density_prev` a fim de realizar a interpolação de dados a partir de esta textura. Os parâmetros são enviados pelo método `calculateNextDensity()` em que `density`, `density_prev`, `u` e `v` são enviados como parâmetros de referência a `tex`, `tex0`, `_u` e `_v` respetivamente, `b` é inicializado com 0 para diferenciar que a tarefa de difusão será realizada com a densidade e não com a velocidade. Esses parâmetros são definidos na Linha 1.

Neste processo usaremos o programa *shader* `addVectShader` que é o responsável por executar os passos da advecção na GPU (Linhas 4-15), fazendo uso dos FBO `density_prev`, `u` e `v` para realizar os cálculos necessários no passo de tempo anterior (representado na Linha 10). Finalmente, utilizamos os resultados gerados pelo `addVectShader` para realizar o *rendering* à textura `tex` (Linha 18) definindo os contornos dessa última por meio do método `sentBnd()` (Linha 26).

Figura 12 – Advecção de textura de densidade: (a) Traçar o *texel* para trás através da velocidade, (b) Interpolar o valor a partir do valor anterior, (c) Atribuir este valor ao *texel* atual.



Fonte – Figura adaptada de (12).

Listagem 7 – Solver.cpp: método `advect()`.

```
1 void Solver::advect(GPUBuffer &tex, GPUBuffer &tex0, int b, GPUBuffer &_u,
      GPUBuffer &_v)
2 {
3   //Bind shader and pass variables as uniforms
4   addVectShader.bind();
5   addVectShader.uniform("tex0", 0);
6   addVectShader.uniform("u", 1);
```

```
7   addVectShader.uniform("v", 2);
8   addVectShader.uniform("OneNStep", OneNStep);
9   addVectShader.uniform("OnePrevOneNStep", 1.0f - OneNStep);
10  addVectShader.uniform("dt", (1.0f - OneNStep) * delta);
11
12  //Bind textures
13  tex0.getTexture().bind(0);
14  _u.getTexture().bind(1);
15  _v.getTexture().bind(2);
16
17  //using the velocity forces to move around the densities
18  render(tex);
19
20  //clean all
21  _u.getTexture().unbind();
22  _v.getTexture().unbind();
23  addVectShader.unbind();
24
25  //manage boundaries
26  setBnd(tex, b);
27 }
```

Fonte – Listagem adaptada de (30).

O arquivo *shader* da advecção executado na GPU é apresentado na Listagem 8. Nele as texturas são representadas mediante o tipos de variável `sampler2D` tanto pra densidade inicial (`tex0`, Linha 1) quanto para as velocidades (`u`, `v` Linhas 2-3). A constante `OnePrevNStep` e usada como delimitador pra cada célula da textura de velocidade. A fim de que os vizinhos de uma célula em estudo não sejam *texels* inexistentes controlamos mediante o condicional `if` na Linha 13. Cabe ressaltar que o cálculo é feito no passo de tempo anterior (`dt`, Linha 6). Nas Linhas 16-23 temos desenvolvida a Equação 11 que foi descrita no capítulo 3. Nas Linhas 16-17 é calculada a célula (*texel*) no passo de tempo anterior através da velocidade. Entre as Linhas 18-21 realizamos o cálculo dos índices dos vizinhos do *texel* no passo de tempo anterior (fazendo uso da constante `OneNStep`). Finalmente na Linha 23 aplicamos a Equação 11 para interpolar e atribuir este resultado ao fragmento mediante a variável `outColor`. Caso seja um *texel* que pertence ao contorno não efetuamos a resolução da equação (Linha 29).

Listagem 8 – *Fragment Shader* de Advecção de Texturas: fadvect.glsl.

```
1  uniform sampler2D tex0;
2  uniform sampler2D u;
3  uniform sampler2D v;
4  uniform float OneNStep;
5  uniform float OnePrevOneStep;
6  uniform float dt;
7
8  in vec2 fcoordText;
9  out vec4 outColor;
10
11 void main()
12 {
13   if (all(greaterThan(fcoordText.xy, vec2(0.0, 0.0))) &&
14     all(lessThan(fcoordText.xy, vec2(1.0, 1.0))))
15   {
```

```
16      vec2 uv = vec2(texture(u, fcoordText.xy).r, texture(v, fcoordText.xy).r
            );
17      vec2 xy = clamp(fcoordText.xy - dt * uv, 0.0, OnePrevOneStep);
18      vec2 val1 = mod(xy, OneNStep);
19      vec2 val0 = vec2(1.0, 1.0) - val1;
20      vec2 ab0 = ((xy / OneNStep) - val1) * OneNStep;
21      vec2 ab1 = ab0 + OneNStep;
22
23      outColor =val0.s * (val0.t * texture(tex0, ab0) +
24            val1.t * texture(tex0, vec2(ab0.x, ab1.y))) +
25            val1.s * (val0.t * texture(tex0, vec2(ab1.x, ab0.y)) +
26            val1.t * texture(tex0, ab1));
27    }
28    else
29      outColor = texture(tex0, fcoordText.xy);
30  }
```

Fonte – Listagem adaptada de (30).

4.3.2 Textura de Velocidade

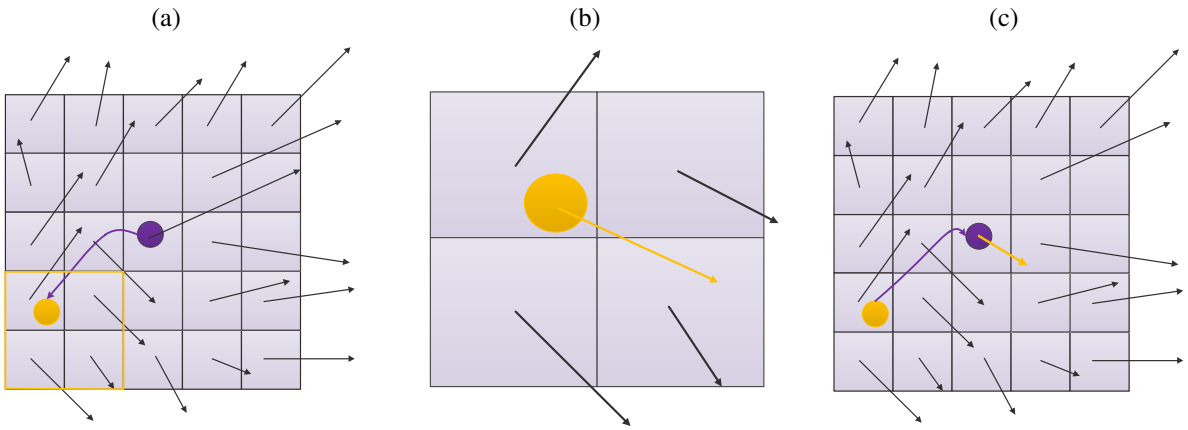
Do mesmo modo em que a textura da densidade atua em três etapas, a textura de velocidade também realiza um comportamento parecido em cada passo de tempo, pois ela muda segundo três fatores:

1. Adição de forças;
2. Difusão viscosa;
3. Auto-advecção (Definida na Seção 3.3.1).

Para descrição do comportamento da textura de velocidade é possível usar as listagens que foram desenvolvidas para a textura de densidade e aplicá-las para atualizar a textura de velocidade para os eixos *x* e *y*, ou seja para os FBO `u`, `v`. O processo de auto-advecção pode ser observado na Figura 13 que, de forma semelhante em que foi descrita a Figura 12, este processo consta de três passos, consistindo em traçar a célula (*texel*) através da velocidade ao passo de tempo anterior com a finalidade de interpolar o valor obtido em razão dos vizinhos mais próximos naquele passo de tempo, para que seguidamente possamos atribuir este valor interpolado ao valor da célula em estudo. A diferença está no fato de que a célula em estudo pertence às texturas de velocidade `u` e `v`.

Baseadas nestas informações, o processo de criação de texturas de velocidade descrito na Listagem 9 mediante o método `calculateNextVelocity()`, resolve os três passos dos itens anteriormente definidos para cada eixo da velocidade (Linhas 4-7,13-14). Os FBO a usar são `u_prev`, `u`, `v_prev` e `v` que representam as velocidades iniciais e finais nos eixos *x* e *y* respetivamente. No caso da difusão e advecção usamos os valores (como parâmetros) 1 e 2 para diferenciar esses processos no cálculo dos contornos, no eixo *x* o valor 1 e no eixo *y* o valor 2. A variável `viscosity` controla a viscosidade da textura de velocidade, ou seja, como será

Figura 13 – Advecção de textura de velocidade: (a) Traçar a posição central da célula para trás, (b) Interpolar a velocidade, (c) Atribuir este valor para a célula de partida.



Fonte – Figura adaptada de (12).

realizado o escoamento das texturas; quanto maior o valor, maior a viscosidade (ver definição na Seção 3.2.2). Como observado nas Linhas 10 e 15, temos um novo método chamado de `project()` que não está presente na criação da textura de densidade. Este método é o responsável da incompressibilidade do fluido (ver definição na Seção 3.2.3), ou seja pela conservação da massa. Usamos duas vezes a projeção porque a rotina da advecção se comporta com mais precisão quando as texturas estão conservando a sua massa.

Listagem 9 – Solver.cpp: método `calculateNextVelocity()`.

```
1  void Solver::calculateNextVelocity()
2  {
3    //Add source and difusion
4    addSource(u, u_prev);
5    addSource(v, v_prev);
6    diffuse(u_prev, u, 1, viscosity);
7    diffuse(v_prev, v, 2, viscosity);
8
9    //mass conservation
10   project(u_prev, v_prev, u, v);
11
12   //Auto-Advect
13   advect(u, u_prev, 1, u_prev, v_prev);
14   advect(v, v_prev, 2, u_prev, v_prev);
15   project(u, v, u_prev, v_prev);
16 }
```

Fonte – Listagem adaptada de (30).

Projeção: A projeção fornece a conservação de massa, comumente apresentada na forma de redemoinhos dentro do fluido. A ideia explicada por Stam é conservar a massa no último passo do processo de atualização da textura da velocidade, mediante a técnica de decomposição de *Helmholtz-Hodge* (15). A seguir vamos descrever este processo mediante a Listagem 10.

O processo da projeção recebe como parâmetros os FBO da textura de velocidade `u_prev`, `v_prev`, `u` e `v`. Além disso, utiliza três programas *shaders* `project1Shader`, `project2Shader` e `project3Shader`, para cada uma das etapas (que chamaremos de *Step*) em que está dividido este processo. Na primeira etapa (*Step1*) descritas nas Linhas 4-23 é usado o `project1Shader` para executar o arquivo GLSL da Listagem 11 a fim de inicializar a textura de velocidade. Na Linha 4 o método `initField()` ativa o FBO `_p`, logo são enviadas as variáveis ao `project1Shader` (Linhas 7-11) e as texturas enviadas são vinculadas nas Linhas 14-15. O resultado feito por este programa é renderizado na Linha 18 ao FBO `_div`. Nesta etapa é realizada a resolução da Equação 12 para a pressão com os resultados que foram renderizados no FBO `_div` obtendo uma nova textura sobre o FBO `_p`, onde $\alpha = 1$ e $\beta = 4$.

A segunda e terceira etapa (*Steps* 2 e 3) realizam o mesmo cálculo sobre as texturas, porém cada uma sobre o eixo x (Linhas 31-50) e y (Linhas 53-72) respectivamente, a fim de obter o campo de incompressibilidade, subtraindo o gradiente `_p` a partir de nossas texturas da velocidade `_u` e `_v`. Nestas duas etapas realizamos a projeção sobre os FBO `_u` (Linha 45) e `_v` (Linha 67) fazendo uso do FBO temporal `tempBuffer` (Linhas 31 e 53).

O processo de projeção finaliza definindo os contornos dos FBO obtidos na segunda e terceira etapa (Linhas 75-76).

Listagem 10 – Solver.cpp: método `project()`.

```
1  void Solver::project(GPUBuffer &_u, GPUBuffer &_v, GPUBuffer &_p, GPUBuffer
       &_div)
2  {
3    //Step1: Init gradient;
4    initField(_p);
5
6    //Bind shader for Step 1 and pass variables as uniforms
7    project1Shader.bind();
8    project1Shader.uniform("u", 0);
9    project1Shader.uniform("v", 1);
10   project1Shader.uniform("OneNStep", OneNStep);
11   project1Shader.uniform("N", (float) N);
12
13   //Bind textures
14   _u.getTexture().bind(0);
15   _v.getTexture().bind(1);
16
17   //Render init values
18   render(_div);
19
20   //clean Step 1
21   _u.getTexture().unbind();
22   _v.getTexture().unbind();
23   project1Shader.unbind();
24
25   //Stam' algorithm
26   setBnd(_div, 0);
27   setBnd(_p, 0);
28   linSolve(_p, _div, 0, 1, 4);
29
30   //Step2: Interpolate values to _u
```

```
31    _u.blitTo(tempBuffer,_u.getBounds(),tempBuffer.getBounds());
32
33    //Bind shader for Step 2 and pass variables as uniforms
34    project2Shader.bind();
35    project2Shader.uniform("u0", 0);
36    project2Shader.uniform("p", 2);
37    project2Shader.uniform("OneNStep", OneNStep);
38    project2Shader.uniform("N", (float) N);
39
40    //Bind texture
41    tempBuffer.getTexture().bind(0);
42    _p.getTexture().bind(2);
43
44    //Render Incompressibility in axis x
45    render(_u);
46
47    //clean Step 2
48    tempBuffer.getTexture().unbind();
49    _p.getTexture().unbind();
50    project2Shader.unbind();
51
52    //Step3: Interpolate values to _v
53    _v.blitTo(tempBuffer,_v.getBounds(),tempBuffer.getBounds());
54
55    //Bind shader for Step 3 and pass variables as uniforms
56    project3Shader.bind();
57    project3Shader.uniform("v0", 1);
58    project3Shader.uniform("p", 2);
59    project3Shader.uniform("OneNStep", OneNStep);
60    project3Shader.uniform("N", (float) N);
61
62    //Bind texture
63    tempBuffer.getTexture().bind(1);
64    _p.getTexture().bind(2);
65
66    //Render Incompressibility in axis y
67    render(_v);
68
69    //clean Step 3
70    tempBuffer.getTexture().unbind();
71    _p.getTexture().unbind();
72    project3Shader.unbind();
73
74    //finish stam' algorithm
75    setBnd(_u, 1);
76    setBnd(_v, 2);
77 }
```

Fonte – Listagem adaptada de (30).

O arquivo *shader* da primeira etapa da projeção executado na GPU é apresentado na Listagem 11. Nele as texturas são representadas mediante o tipos de variável `sampler2D` tanto para velocidade em x (`u`, Linha 1) quanto para velocidade em y (`v` Linhas 2). `OneNStep` é utilizado como delimitador da nossa grade de tamanho `N`. Na Linha 11 é realizada a inicialização *texel* por *texel* do FBO `_div` descrito na Listagem 10, a partir dos dados iniciais das velocidades em ambos os eixos. Este cálculo é enviado ao fragmento por meio da variável `outColor`.

Listagem 11 – *Fragment Shader* de projeção (*Step1*): fproject1.glsl.

```
1  uniform sampler2D u;
2  uniform sampler2D v;
3  uniform float OneNStep;
4  uniform float N;
5
6  in vec2 fcoordText;
7  out vec4 outColor;
8
9  void main()
10 {
11   outColor = vec4(-0.5, 0.0, 0.0, 0.0) *
12         (texture(u, fcoordText.xy + vec2( OneNStep, 0.0)) -
13         texture(u, fcoordText.xy + vec2(-OneNStep, 0.0)) +
14         texture(v, fcoordText.xy + vec2(0.0,  OneNStep)) -
15         texture(v, fcoordText.xy + vec2(0.0, -OneNStep))) / N;
16 }
```

Fonte – Listagem adaptada de (30).

O arquivo *shader* da segunda etapa da projeção executado na GPU é apresentado na Listagem 12. Nele as texturas são representadas mediante o tipos de variável `sampler2D` tanto para a velocidade em *x* (`u0`, Linha 1) quanto para o gradiente (`p` Linha 2). `OneNStep` é utilizado como delimitador da nossa grade de tamanho `N`. Na Linha 11 é realizado o cálculo da incompressibilidade no eixo *x texel* por *texel*. Este cálculo é enviado ao fragmento por meio da variável `outColor`.

Listagem 12 – *Fragment Shader* de projeção (*Step2*): fproject2.glsl.

```
1  uniform sampler2D u0;
2  uniform sampler2D p;
3  uniform float OneNStep;
4  uniform float N;
5
6  in vec2 fcoordText;
7  out vec4 outColor;
8
9  void main()
10 {
11   outColor = texture(u0, fcoordText.xy) -
12         (vec4(0.5, 0.0, 0.0, 0.0) * N *
13         (texture(p, fcoordText.xy + vec2( OneNStep, 0.0)) -
14         texture(p, fcoordText.xy + vec2(-OneNStep, 0.0))));
15 }
```

Fonte – Listagem adaptada de (30).

O arquivo *shader* da terceira etapa da projeção executado na GPU é apresentado na Listagem 13. Nele as texturas são representadas mediante o tipos de variável `sampler2D` tanto para a velocidade em *y* (`v0`, Linha 1) quanto para o gradiente (`p` Linha 2). `OneNStep` é utilizado como delimitador da nossa grade de tamanho `N`. Na Linha 11 é realizado o cálculo da incompressibilidade no eixo *y texel* por *texel*. Este cálculo é enviado ao fragmento por meio da variável `outColor`.

Listagem 13 – *Fragment Shader* de projeção (*Step3*): fproject3.glsl.

```
uniform sampler2D v0;
uniform sampler2D p;
uniform float OneNStep;
uniform float N;

in vec2 fcoordText;
out vec4 outColor;

void main()
{
  outColor = texture(v0, fcoordText.xy) -
        (vec4(0.5, 0.0, 0.0, 0.0) * N *
        (texture(p, fcoordText.xy + vec2(0.0,  OneNStep)) -
        texture(p, fcoordText.xy + vec2(0.0, -OneNStep))));
}
```

Fonte – Listagem adaptada de (30).

4.3.3 Região de Contornos

Como explicado na Seção 3.4.2, vamos a definir as regiões (Listagem 14) da fronteira como delimitadores de nossa simulação em que as tintas digitais refletem quando colidirem nos contornos. Na Listagem 14 são enviados os dados através do programa *shader* `setBndShader` (Linhas 8-13) e a textura é vinculada na Linha 14. A cópia de dados ao FBO `tempBuffer` é realizada na Linha 5 por meio da função `blitTo()` da biblioteca Cinder, a fim de usá-lo como FBO auxiliar nos cálculos realizados na GPU. A variável `b` controla o eixo em que é realizado o processo de criação de contornos.

Listagem 14 – Solver.cpp: método `setBnd()`.

```
void Solver::setBnd(GPUBuffer &x, int b)
{
  //copy x to tempBuffer: copy a block of pixels from
  //the read framebuffer to the draw framebuffer
  x.blitTo(tempBuffer,x.getBounds(),tempBuffer.getBounds());

  //Bind shader and pass variables as uniforms
  setBndShader.bind();
  setBndShader.uniform("x", 0);
  setBndShader.uniform("b", b);
  setBndShader.uniform("OneNStep", OneNStep);

  //Bind textures
  tempBuffer.getTexture().bind(0);

  //setBnd(x, b);
  render(x);

  //clean all
  tempBuffer.getTexture().unbind();
  setBndShader.unbind();
}
```

Fonte – Listagem adaptada de (30).

O arquivo *shader* da região de contornos executado na GPU é apresentado na Listagem 15. Nele a textura `x` é representada mediante o tipo de variável `sampler2D`. `OneNStep` é utilizado como delimitador e a variável `b` controla o eixo, como explicado na listagem anterior ($x = 1$, $y = 2$). São definidos os contornos da seguinte maneira: vertical esquerda (Linha 14), vertical direita (Linha 18), horizontal superior (Linha 22) e horizontal inferior (Linha 26). Para os cantos temos o cálculo da média dos dois valores mais próximos (ver exemplo na Equação 5) desenvolvido nas Linhas 32, 34, 41 e 43. Este cálculo é enviado ao fragmento por meio da variável `outColor`.

Listagem 15 – *Fragment Shader* de Região de Contornos: fsetbnd.glsl.

```
1  uniform sampler2D x;
2  uniform int b;
3  uniform float OneNStep;
4
5  in vec2 fcoordText;
6  out vec4 outColor;
7
8  void main()
9  {
10   outColor = texture(x, fcoordText.xy);
11
12   //Left vertical
13   if (fcoordText.x == 0.0 && fcoordText.y > 0.0 && fcoordText.y < 1.0)
14     outColor = (b == 1) ? -texture(x, vec2(OneNStep, fcoordText.y)) :
         texture(x, vec2(OneNStep, fcoordText.y));
15
16   //Right vertical
17   else if (fcoordText.x == 1.0 && fcoordText.y > 0.0 && fcoordText.y < 1.0)
18     outColor = (b == 1) ? -texture(x, vec2(1.0 - OneNStep, fcoordText.y)) :
          texture(x, vec2(1.0 - OneNStep, fcoordText.y));
19
20   //Top horizontal
21   if (fcoordText.y == 0.0 && fcoordText.x > 0.0 && fcoordText.x < 1.0)
22     outColor = (b == 2) ? -texture(x, vec2(fcoordText.x, OneNStep)) :
         texture(x, vec2(fcoordText.x, OneNStep));
23
24   //Bottom horizontal
25   else if (fcoordText.y == 1.0  && fcoordText.x > 0.0 && fcoordText.x <
       1.0)
26     outColor = (b == 2) ? -texture(x, vec2(fcoordText.x, 1.0 - OneNStep)) :
          texture(x, vec2(fcoordText.x, 1.0 - OneNStep));
27
28   //Bottom right corner and Top right corner
29   if (fcoordText.x == 0.0)
30   {
31     if (fcoordText.y == 0.0)
32       outColor = 0.5 * (texture(x, vec2(OneNStep, 0.0)) + texture(x, vec2
           (0.0, OneNStep)));
33     else if (fcoordText.y == 1.0)
34       outColor == 0.5 * (texture(x, vec2(OneNStep, 1.0)) + texture(x, vec2
           (0.0, 1.0 - OneNStep)));
35   }
36
37   //Bottom left corner and Top left corner
38   else if (fcoordText.x == 1.0)
39   {
40     if (fcoordText.y == 0.0)
```

```
41        outColor = 0.5 * (texture(x, vec2(1.0 - OneNStep, 0)) + texture(x,
              vec2(1.0, OneNStep)));
42      else if (fcoordText.y == 1.0)
43        outColor = 0.5 * (texture(x, vec2(1.0 - OneNStep, 1.0)) + texture(x,
              vec2(1.0, 1.0 - OneNStep)));
44    }
45  }
```

Fonte – Listagem adaptada de (30).

4.3.4 Área Circular

No marmoreio em papel as tintas criadas sobre a superfície aquosa tem aparência circular. Para atingir esse tipo de simulação é preciso criar texturas de densidades circulares. Para isso, apresentamos a Listagem 16 que cria uma textura circular através do método `addDensity()` da Linha 1, que é invocado pela classe `OpenGLWidget`. As deformações das tintas são realizadas por meio da textura da velocidade, usando o método `addVelocity()` da Linha 6, que é invocado pela classe `OpenGLWidget`.

As variáveis usadas por ambos os métodos são muitos similares. A posição quando o usuário pressiona sobre a tela é enviada pelas variáveis `x` e `y`. O raio `r` define a circunferência, e a intensidade de movimento com que interage o usuário é controlada pela variável `s`. Os dados destes métodos são enviados à GPU mediante o método `addForceArea()` (Linhas 12-35).

O método `addForceArea()` envia os dados, das variáveis anteriormente ditas, ao programa *shader* `addForceShader`. Na Linha 16 é realizada a cópia de dados ao FBO `tempBuffer` por meio da função `blitTo()` da biblioteca Cinder, a fim de usá-lo como FBO auxiliar nos cálculos realizados na GPU. Nas Linhas 21, 22 e 24 realizamos a normalização dos dados ao nosso domínio. A textura é vinculada na Linha 27. Os dados obtidos pelo *shader* são renderizados na Linha 30.

Listagem 16 – Solver.cpp: métodos `addDensity()`, `addVelocity()` e `addForceArea()`.

```
1  void Solver::addDensity(unsigned int x, unsigned int y, float s, float r,
       int layer)
2  {
3    addForceArea(density_prev[layer], x, y, s, r);
4  }
5
6  void Solver::addVelocity(unsigned int x, unsigned int y, float sx, float sy
       , float r)
7  {
8    addForceArea(u_prev, x, y, sx, r);
9    addForceArea(v_prev, x, y, sy, r);
10 }
11
12 void Solver::addForceArea(GPUBuffer &field, const int x, const int y, float
       s, float r)
13 {
14   //copy field to tempBuffer: copy a block of pixels from
15   //the read framebuffer to the draw framebuffer
16   field.blitTo(tempBuffer,field.getBounds(),tempBuffer.getBounds());
17
```

```
18    //Bind shader and pass variables as uniforms
19    addForceShader.bind();
20    addForceShader.uniform("field0", 0);
21    addForceShader.uniform("x_center", (float) x * OneNStep);
22    addForceShader.uniform("y_center", (float) y * OneNStep);
23    addForceShader.uniform("s", s);
24    addForceShader.uniform("r", r * OneNStep);
25
26    //Bind texture
27    tempBuffer.getTexture().bind(0);
28
29    //perform the actual force adding calculation
30    render(field);
31
32    //clean all
33    tempBuffer.getTexture().unbind();
34    addForceShader.unbind();
35  }
```

Fonte – Listagem adaptada de (30).

O arquivo *shader* da criação de áreas circulares executado na GPU é apresentado na Listagem 17. Nele a textura `field0` é representada mediante o tipo de variável `sampler2D`. As variáveis restantes são definidas de tipo `float` (Linhas 2-5). As coordenadas de textura são controladas por meio da variável `fcoordText` Os resultados deste processo são enviados ao fragmento mediante a variável `outColor`. Nas Linhas 16-19 é realizada a verificação do `texel`; ele deve estar dentro de uma região ao redor do raio. Esta verificação é feita em cascata para um maior desempenho. Nas Linhas 20-23 é feita a criação da área de textura circular com centro (x_center, y_center).

Listagem 17 – *Fragment Shader* de criação de Área Circular: faddforce.glsl.

```
1  uniform sampler2D field0;
2  uniform float x_center;
3  uniform float y_center;
4  uniform float s;
5  uniform float r;
6  in vec2 fcoordText;
7  out vec4 outColor;
8
9  void main(){
10   vec4 SourceTexel = texture(field0, fcoordText.xy);
11   outColor = SourceTexel;
12   float cur_x     = fcoordText.x;
13   float cur_y     = fcoordText.y;
14
15   //check, if the current pixel is within a bounding box around the radius
16   if (cur_x >= clamp(x_center - r, 0.0, 1.0))
17     if (cur_x <= clamp(x_center + r, 0.0, 1.0))
18       if (cur_y >= clamp(y_center - r, 0.0, 1.0))
19         if (cur_y <= clamp(y_center + r, 0.0, 1.0)){
20           float dx = x_center - cur_x;
21           float dy = y_center - cur_y;
22           float f  = 1.0 - (sqrt(dx * dx + dy * dy) / r);
23           outColor = SourceTexel + (clamp(f, 0.0, 1.0) * s);
24         }
25 }
```

Fonte – Listagem adaptada de (30).

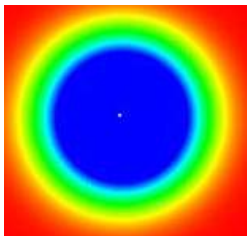
4.4 Simulação de Camadas

Na Seção 4.3.1 temos desenvolvido a simulação de nossas texturas para uma cor apenas, pelo qual precisamos implementar as diferentes cores que os usuários vão adicionar para criar os padrões resultantes. Para atingir este objetivo e implementá-lo em nosso protótipo é necessário definir o gradiente das cores que serão geradas na memória RAM (CPU) para após serem copiadas em nossa memória VRAM (GPU). Este processo foi feito seguindo o *skate* de Phagor (28).

4.4.1 Gradiente de cores

A ideia principal é gerar um gradiente de cores no sentido de apresentar aspecto em 3D nas tintas de marmoreio digital. Um gradiente de cores é uma sequencia de tons contínuos, ou seja, eles são sobrepostos, cada um formando uma transição suave entre as cores. Em nosso protótipo é visível de forma que cada gradiente representa uma cor específica (Figura 14).

Figura 14 – Gradiente de cores: Visualização de cinco gradientes (azul, azul claro, verde, amarelo e laranja) renderizadas na GPU.



Para atingir esta ideia vamos apresentar os principais métodos que descrevem o processo. Como primeiro passo deve-se gerar os nós que armazenaram o gradiente na CPU. Este processo apresentado na Listagem 18 é controlado pelo método `makeDefaultGradient()` que tem como parâmetros a cor a gerar (`colorVal`) e se o gradiente a gerar será padrão ou pela escolha do usuário (`defaultgradient`. A cor é normalizada nas Linhas 6-8 para ser processada internamente como `ColorA` (Linha 14, 15, 18 e 19). Os casos do `switch` representam a cor padrão (`case cBLUE`) e a cor escolhida pelo usuário (`case cSELECT`). O método `addNode()` permite alocar (Linha 26) dois nós na CPU, um deles é a cor preta (cor de fundo da tela) e a outra é a cor escolhida pelo usuário.

Listagem 18 – Gradiente.cpp: métodos `makeDefaultGradient()` e `addNode()`.

```
1 void Gradient::makeDefaultGradient(int defaultgradient, QColor colorVal)
2 {
3   clear();
4
5   //The QColor class returns values between 0-255. To normalize them to 0-1
6   float R = ((float) colorVal.red()) / 255.0;
7   float G = ((float) colorVal.green()) / 255.0;
8   float B = ((float) colorVal.blue()) / 255.0;
```

```
9
10    //The color blue is a default color of the system
11    switch (defaultgradient)
12    {
13      case cBLUE:
14        addNode(0, ColorA(0, 0, 0, 1));
15        addNode(1, ColorA(0, 0, 1, 1));
16        break;
17      case cSELECT:
18        addNode(0, ColorA(0, 0, 0, 1));
19        addNode(1, ColorA(R, G, B, 1));
20        break;
21    }
22  }
23
24  void Gradient::addNode(float location, const ColorA& c)
25  {
26    nodes.push_back(Gradient::GradientNode(location, c));
27
28    //Locations are sorted ascending
29    std::sort(nodes.begin(), nodes.end());
30  }
```

Fonte – Listagem adaptada de (28).

Como segundo passo deve-se copiar os nós gerados na etapa anterior dentro de um vetor de cores. O método `fillVectorOfColors()` na Listagem 19 permite alocar num vetor (`colors`) os tons que compõem o gradiente padrão ou aquele escolhido pelo usuário, ou seja cada um dos tons formando uma transição suave entre as cores, para isto utiliza o método `getColor()` (Linha 9). A quantidade de tons é controlada pelo parâmetro `numberofcolors`, que segundo a experiência do autor (28) deve ser igual a 2048. O método `getColor()` gera os 2048 tons que compõem nosso gradiente com valores a partir de 0 até 1 (Linhas 22-29). Na Linha 18 é controlado essa quantidade a gerar. Na Linha 20 o condicional `if` verifica que o nó seja alocado numa posição existente.

Listagem 19 – Gradiente.cpp: métodos `getColor()` e `fillVectorOfColors()`.

```
1  void Gradient::fillVectorOfColors(int numberofcolors, ColorVector& colors)
2  {
3    ColorA current_color;
4    colors.clear();
5    colors.reserve(numberofcolors);
6
7    for (int i=0; i <numberofcolors; i++)
8    {
9      getColor((float) i / (float) (numberofcolors - 1), current_color);
10     colors.push_back(current_color);
11   }
12 }
13
14 void Gradient::getColor(float location, ColorA& col)
15 {
16   float Band0, Band1, ScaleFactor;
17
18   for (int c=0; c < nodes.size() - 1; c++)
19   {
20     if (location >= nodes[c].location && location <= nodes[c+1].location)
```

```
21      {
22        Band0 = nodes[c+1].location - nodes[c].location;
23        Band1 = location - nodes[c].location;
24        ScaleFactor = Band1 / Band0;
25
26        col.r = ScaleFactor*(nodes[c+1].col.r - nodes[c].col.r)+ nodes[c].col
              .r;
27        col.g = ScaleFactor*(nodes[c+1].col.g - nodes[c].col.g)+ nodes[c].col
              .g;
28        col.b = ScaleFactor*(nodes[c+1].col.b - nodes[c].col.b)+ nodes[c].col
              .b;
29        col.a = ScaleFactor*(nodes[c+1].col.a - nodes[c].col.a)+ nodes[c].col
              .a;
30        return;
31      }
32    }
33    col.r = 0;
34    col.g = 0;
35    col.b = 0;
36    col.a = 0;
37  }
```

Fonte – Listagem adaptada de (28).

Como terceiro e último passo deve-se realizar a transferência de dados à GPU. Para isto, utilizamos o método `initGradient()` da Listagem 20. Nela é descrita a copia de gradientes da CPU para GPU. Toda vez que o vetor gerado pelo gradiente está alocado na memória RAM (Linhas 3 e 4) é necessário copiá-lo pra memória VRAM pixel a pixel (Linhas 8 e 9), mediante o uso de texturas (Linha 12). A textura usada na CPU é de tipo `Surface32f` da biblioteca Cinder, que é uma representação na memória de uma imagem. A textura usada na GPU é de tipo `gl::Texture` também da biblioteca Cinder que representa uma textura em OpenGL.

Listagem 20 – OpenGLWidget.cpp: método `initGradient()`.

```
1  void OpenGLWidget::initGradient(int colorscheme, int layer)
2  {
3    GradientMaker.makeDefaultGradient(colorscheme,colorVal);
4    GradientMaker.fillVectorOfColors(cGrandientCountColor, GradientColors_GPU
       );
5
6    Surface32f paintedgradient = Surface32f(cGrandientCountColor, 1, false,
       SurfaceChannelOrder::RGBA);
7    Vec2i coord = Vec2i(0, 0);
8    for (coord.x = 0; coord.x < cGrandientCountColor; coord.x++)
9      paintedgradient.setPixel(coord, GradientColors_GPU[coord.x]);
10
11   //copy RAM to VRAM
12   GradientVRAM[layer] = gl::Texture(paintedgradient);
13 }
```

Fonte – Listagem adaptada de (30).

4.4.2 Densidade Multi-camada

O conceito de multi-camadas foi incorporado devido ao fato que no marmoreio de papel é usada uma solução chamada de efeito *ox gall*, com a finalidade de prevenir que se misturem as

diferentes cores adicionadas pelos artistas. Nesse caso, desenvolvemos o conceito de densidade multi-camada.

A densidade multi-camada esta composta por um número de camadas entre 8 e 20, pois um número menor do que isto não oferece diversidade na criação de padrões e um número maior impacta no desempenho de nosso protótipo (simulação em tempo real). Cada camada representa um gradiente (cor) aplicado pelos usuários.

Na Listagem 21 está descrito o desenvolvimento do desenho das texturas de densidade multi-camada, onde cada cor gerada pela nossa classe `Gradiente` é armazenada na textura `GradientVRAM[]`; esta textura é relacionada a cada camada da densidade `densityToDraw[]` (Linha 6). As densidades foram obtidas mediante o processo descrito na Seção 4.3.1. A macro `QUANTLAYER` controla a quantidade de camadas a serem inseridas pelos usuários. Segundo Khronos Inc. (31), na indexação em matrizes `uniform` para imagens em *shaders* são usadas apenas expressões integrais constantes, portanto nas Linhas 13-22 é obrigatório inserir uma a uma as unidades da textura de tipo `const int`, que foram definidas nas Linhas 9-10. Para fins didáticos apenas são apresentadas 5 camadas na Listagem 21 que faz uso do programa *shader* `drawCinderShader` que é de tipo `GlslProg` da biblioteca Cinder, para envio dos dados à GPU. Nas Linhas 23-24 são usados valores estabelecidos pelo autor (28) com a finalidade de dar brilho às densidades. O brilho é ativado mediante a variável booleana `lighting`. Os *pixels* são normalizados ao nosso domínio na variável `OnePixel`. A variável `destSize` obtém a posição do toque realizado pelo usuário. Nas Linhas 30-34 são vinculadas as texturas. O *rendering* sobre a tela é realizado na Linha 36 mediante a função `glDrawArrays()` do OpenGL.

Listagem 21 – OpenGLWidget.cpp: método `drawFluidLayer()`.

```
1  void OpenGLWidget::drawFluidLayer()
2  {
3      makeCurrent();
4
5    for(unsigned int i=0;i<QUANTLAYER;i++)
6      densityToDraw[i] = solver.getTextureDensity(i);
7
8    //Texture unit
9    const int TexU_Density[] = {0,1,2,3,4};
10   const int TexU_GradientColors[] = {5,6,7,8,9};
11
12   drawCinderShader.bind();
13   drawCinderShader.uniform("Density[0]", TexU_Density[0]);
14   drawCinderShader.uniform("Density[1]", TexU_Density[1]);
15   drawCinderShader.uniform("Density[2]", TexU_Density[2]);
16   drawCinderShader.uniform("Density[3]", TexU_Density[3]);
17   drawCinderShader.uniform("Density[4]", TexU_Density[4]);
18   drawCinderShader.uniform("GradientColors[0]", TexU_GradientColors[0]);
19   drawCinderShader.uniform("GradientColors[1]", TexU_GradientColors[1]);
20   drawCinderShader.uniform("GradientColors[2]", TexU_GradientColors[2]);
21   drawCinderShader.uniform("GradientColors[3]", TexU_GradientColors[3]);
22   drawCinderShader.uniform("GradientColors[4]", TexU_GradientColors[4]);
23   drawCinderShader.uniform("LightingFactor", lightingFac / 256.0f);
24   drawCinderShader.uniform("LightingPower", lightingPow);
25   drawCinderShader.uniform("OnePixel", OnePixel);
```

```
26    drawCinderShader.uniform("lighting", lighting);
27    drawCinderShader.uniform("destSize", lighting);
28
29    //bind
30    for(unsigned int i=0;i<QUANTLAYER;i++)
31    {
32      densityToDraw[i].bind(TexU_Density[i]);
33      GradientVRAM[i].bind(TexU_GradientColors[i]);
34    }
35
36    createVBOs();
37
38    //unbind
39    for(unsigned int i=0;i<QUANTLAYER;i++)
40    {
41      densityToDraw[i].unbind();
42      GradientVRAM[i].unbind();
43    }
44
45    drawCinderShader.unbind();
46
47    update();
48  }
```

O arquivo *shader* da densidade multi-camada executado na GPU é apresentado na Listagem 22. Nele as texturas `Density[]` e `GradientColors[]` são representadas mediante o tipo de variável `sampler2D` (Linhas 2-3). A variáveis utilizadas foram descritas na Listagem anterior. As coordenadas de textura são controladas por meio da variável `fcoordText`. Os resultados deste processo são enviados ao fragmento mediante a variável `outColor`. Os valores da densidade foram armazenados no canal R (RGB) das texturas; na Linha 25 obtemos esses valores na variável `densityvalue[]`. As coordenadas utilizadas pela textura `GradientColors[]` estão relacionadas com a densidade na Linha 30 através da variável `colorPosition[]`, para enfim se obter a cor na Linha 35 para cada *texel*. O condicional `if` da Linha 38 controla o efeito de brilho sobre as tintas digitais (28). As camadas são geradas nas Linhas 54-58 em que a ideia é adicionar tintas na última camada criada. A variável `treshold` controla a presença da cor (gradiente) no *pixel*.

Listagem 22 – *Fragment Shader* para desenhar as texturas: fdraw.glsl.

```
1  //Input textures
2  uniform sampler2D Density[5];
3  uniform sampler2D GradientColors[5];
4
5  //values of phagor's gradient
6  uniform float LightingFactor;
7  uniform float LightingPower;
8  uniform bool lighting;
9
10 //width/height of the quadratic density texture in pixels
11 uniform float OnePixel;
12
13 //input/output variable
14 in vec2 fcoordText;
15 out vec4 outColor;
16
```

```
17 //lighting calculation
18 float az, bz, nz, w;
19
20 void main()
21 {
22   //the density texture uses only the red channel to store the density
        values
23   float densityvalue[5];
24   for (int i=0; i<5;i++)
25     densityvalue[i] = texture(Density[i], fcoordText).r;
26
27   //clamp/range densityvalue between 0..1 to get the color index inside the
        gradient's texture
28   vec2 colorposition[5];
29   for (int i=0; i<5;i++)
30     colorposition[i] = vec2(clamp(densityvalue[i], 0.0, 1.0), 0.0);
31
32   //Apply color to density
33   vec4 color[5];
34   for (int i=0; i<5;i++)
35     color[i] = texture(GradientColors[i], colorposition[i]);
36
37   //brighten current pixel's color according to a virtual light source
38   if (lighting)
39   {
40     for (int i=0; i<5;i++)
41     {
42       az=densityvalue[i]-texture(Density[i],fcoordText+vec2(OnePixel,0.0)).
           r;
43       bz=densityvalue[i]-texture(Density[i],fcoordText+vec2(0.0,OnePixel)).
           r;
44       nz=1.0 / length(vec3(az, bz, 1.0));
45       w=(colorposition[i].x * LightingFactor) * pow(nz, LightingPower);
46       color[i] += vec4(w, w, w, 0.0);
47       color[i] = min(color[i], 1.0);
48     }
49   }
50
51   //Layer composition
52   float treshold = 0.5;
53
54   if (colorposition[4].x > treshold)  outColor.rgb = color[4].rgb;
55   else if (colorposition[3].x > treshold)  outColor.rgb = color[3].rgb;
56   else if (colorposition[2].x > treshold)  outColor.rgb = color[2].rgb;
57   else if (colorposition[1].x > treshold)  outColor.rgb = color[1].rgb;
58   else outColor.rgb = color[0].rgb;
59   outColor.a = 1.0;
60 }
```

Fonte – Listagem adaptada de (30).

4.5 Nitidez de Texturas

No processo de criação de texturas mediante a simulação de fluidos, as texturas sofrem de borramentos nos contornos porque a natureza iterativa do solucionador (método *Gauss-Seidel* da Seção 3.4) torna a dissipação inevitável. Portanto para melhorar a qualidade das texturas temos aplicado duas técnicas: *Shock Filter* e *Multisample anti-aliasing* (MSAA).

4.5.1 *Shock Filter*

O *Shock Filter* é um filtro utilizado para remover borrões de uma imagem devido ao fato que transforma as transições suaves resultantes da interpolação das texturas em transições abruptas. A teoria matemática deste filtro foi apresentada por Osher & Rudin (32), em que o principio deste método esta baseado na difusão de energia entre pixeis vizinhos; para isto nas áreas da imagem em que a segunda derivada é positiva, as cores dos pixeis são difundidas na direção do gradiente inverso e vice-versa. Para afiar uma imagem corretamente, muitas passagens são normalmente necessárias.

A Listagem 23 apresenta entre as Linhas 25-40 o desenvolvimento do *Shock Filter* sobre o aquivo *shader* implementado na Listagem 22. O *Shock Filter* está adaptado do *shader* implementado pela NVIDIA (33) na linguagem Cg. Este *shader* usa diferenças centradas para estimar o gradiente da imagem e um método de cinco amostras (Linhas 28-32) para determinar a convexidade (o sinal da segunda derivada, Linha 33). Outra mudança realizada em relação à Listagem 22 está apresentada nas Linhas 55-57, devido a que nessas linhas é aplicado o filtro que anteriormente não estava implementado.

Listagem 23 – *Fragment Shader* para remover borrões nas texturas: fdraw.glsl.

```
1  //Input textures
2  uniform sampler2D Density[5];
3  uniform sampler2D GradientColors[5];
4
5  //values of phagor's gradient
6  uniform float LightingFactor;
7  uniform float LightingPower;
8  uniform bool lighting;
9
10 //Touch position
11 uniform vec2 destSize;
12
13 //width/height of the quadratic density texture in pixels
14 uniform float OnePixel;
15
16 //input/output variable
17 in vec2 fcoordText;
18 out vec4 outColor;
19
20 //lighting calculation
21 float az, bz, nz, w;
22
23 //Shock filter Function
24 vec4 Shock_Filter(in sampler2D Image, in vec4 inTex)
25 {
26   float shockMagnitude = 5.9;
27   vec3 inc = vec3(1.0/destSize, 0.0);
28   vec4 curCol = texture(Image, vec2(inTex));
29   vec4 upCol = texture(Image, vec2(inTex) + inc.zy);
30   vec4 downCol = texture(Image, vec2(inTex) - inc.zy);
31   vec4 rightCol = texture(Image, vec2(inTex) + inc.xz);
32   vec4 leftCol = texture(Image, vec2(inTex) - inc.xz);
33   vec4 Convexity = 4.0 * curCol - rightCol - leftCol - upCol - downCol;
34   vec2 diffusion = vec2(dot((rightCol - leftCol) * Convexity, vec4( 1.0,
```

```
      1.0, 1.0, 1.0)), dot((upCol - downCol) * Convexity, vec4( 1.0, 1.0,
      1.0, 1.0)));
35
36   diffusion *= shockMagnitude/(length(diffusion) + 0.00001);
37   curCol += (diffusion.x > 0.0 ? diffusion.x * rightCol : -diffusion.x*
       leftCol) + (diffusion.y > 0.0 ? diffusion.y * upCol : -diffusion.y *
       downCol);
38
39   return curCol/(1.0 + dot(abs(diffusion), vec2(1.0, 1.0)));
40 }
41
42 void main()
43 {
44   //the density texture uses only the red channel to store the density
       values
45   float densityvalue[5];
46   for (int i=0; i<5;i++)
47     densityvalue[i] = texture(Density[i], fcoordText).r;
48
49   //clamp/range densityvalue between 0..1 to get the color index inside the
        gradient's texture
50   vec2 colorposition[5];
51   for (int i=0; i<5;i++)
52     colorposition[i] = vec2(clamp(densityvalue[i], 0.0, 1.0), 0.0);
53
54   //Shock Filter: A Method for Deblurring Images
55   vec4 color[5];
56   for (int i=0; i<5;i++)
57     color[i] = Shock_Filter(GradientColors[i], vec4(colorposition[i].x,
         colorposition[i].y,0.0,0.0));
58
59   //brighten current pixel's color according to a virtual light source
60   if (lighting)
61   {
62     for (int i=0; i<5;i++)
63     {
64       az=densityvalue[i]-texture(Density[i],fcoordText+vec2(OnePixel, 0.0))
           .r;
65       bz=densityvalue[i]-texture(Density[i],fcoordText+vec2(0.0, OnePixel))
           .r;
66       nz=1.0 / length(vec3(az, bz, 1.0));
67       w=(colorposition[i].x * LightingFactor) * pow(nz, LightingPower);
68       color[i] += vec4(w, w, w, 0.0);
69       color[i] = min(color[i], 1.0);
70     }
71   }
72
73   //Layer composition
74   float treshold = 0.5;
75
76   if (colorposition[4].x > treshold)  outColor.rgb = color[4].rgb;
77   else if (colorposition[3].x > treshold)  outColor.rgb = color[3].rgb;
78   else if (colorposition[2].x > treshold)  outColor.rgb = color[2].rgb;
79   else if (colorposition[1].x > treshold)  outColor.rgb = color[1].rgb;
80   else outColor.rgb = color[0].rgb;
81   outColor.a = 1.0;
82 }
```

Fonte – Listagem adaptada de (28) e (33).

4.5.2 *Multisample anti-aliasing*

O anti-*aliasing* é um método de redução do efeito de serra (conhecido como *aliasing*) que é gerado ao desenhar uma reta diagonal num computador, pois a divisão minima da tela é de *pixels*, surgindo este efeito ao longo da reta desenhada. O *Multisample anti-aliasing* é um tipo de anti-*aliasing* para melhorar a qualidade da imagem, em nosso caso, são texturas.

Na Listagem 24 é descrito o desenvolvimento do *Multisample anti-aliasing* (Linhas 8-13). Este processo é desenvolvido mediante a Classe `QSurfaceFormat`[6] de Qt, que representa o formato de um `QSurface`. Este `QSurface`[7] é uma abstração de superfícies renderizáveis em Qt. O formato contém:

- Versão de OpenGL para renderização (Linha 9);
- Tamanho do *deph buffer* (Linha 10);
- Número de *sample* por pixel para o *Multisample* (Linha 11);
- Perfil de OpenGL para renderização (Linha 12).

Estas propriedades são estabelecidas mediante o método `setDefaultFormat()` na Linha 13.

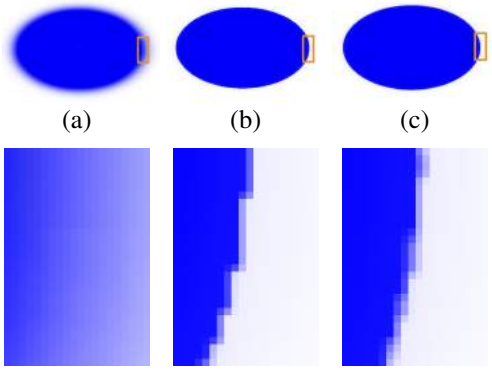
Listagem 24 – main.cpp.

```
1  #include "mainwindow.h"
2  #include <QApplication>
3  #include <QSurfaceFormat>
4
5  int main(int argc, char *argv[])
6  {
7    //MSAA
8    QSurfaceFormat format;
9    format.setVersion(5,2);
10   format.setDepthBufferSize(24);
11   format.setSamples(32);
12   format.setProfile(QSurfaceFormat::CompatibilityProfile);
13   QSurfaceFormat::setDefaultFormat(format);
14
15   QApplication a(argc, argv);
16   MainWindow firstForm;
17   firstForm.showMaximized();
18
19   return a.exec();
20 }
```

6 <http://doc.qt.io/qt-5/qsurfaceformat.html#details> [Online; accessed 2016-12-21]
7 <http://doc.qt.io/qt-5/qsurface.html#details> [Online; accessed 2016-12-21]

Finalmente, os resultados da aplicação destas técnicas são apresentados na Figura 15.

Figura 15 – Nitidez das tintas: A linha superior apresenta uma gota inicial de tinta azul e a linha inferior apresenta um zoom no contorno da gota. (A) Gota inicial da tinta, (b) Tinta com *shock filter*, (c) tinta com *shock filter* e anti-*aliasing*.



4.6 Padrões de Marmoreio

Os padrões de marmoreio digital são os resultados dos movimento das texturas, realizado pelos usuários mediante as ferramentas agulha e pente do marmoreio em papel. Estas ferramentas permitem ou fazer um movimento por vez (agulha) ou vários ao mesmo tempo sobre as tintas (pente).

4.6.1 Movimento de Agulha

Na Listagem 25 é apresentado o movimento de agulha (uma tinta por vez) e tem como entrada de dados a posição em que o usuário vai realizar o movimento (deformação da textura). Nas Linha 3-4 temos o cálculo da intensidade em que o usuário vai realizar o movimento da tinta na tela; se ele pressiona com maior intensidade na tela; maior será o movimento realizado sobre a tinta. Nas Linhas 7-11 é normalizada a posição da tela no domínio das texturas. Entre as Linhas 14-18 são adicionadas as tintas sobre a tela na camada respectiva. O processo é invocado mediante o método `addVelocity()` da classe `Solver` com o valor de `brushVelocity = 0.7f`, que representa o tamanho da agulha sobre as tintas; quanto menor o tamanho, menor a espessura da agulha. Um exemplo de uso deste movimento pode ser observado na Figura 16.

Figura 16 – Padrão obtido com a agulha.

Fonte – (34)

Listagem 25 – OenGLWdget.cpp: método `applyPoint()`.

```
1  void OpenGLWidget::applyPoint(QTouchEvent::TouchPoint &p0, int value)
2  {
3    QPoint TouchPos = p0.pos().toPoint();
4    QPoint TouchVel = TouchPos - TouchLastPos;
5
6    // Convert motion coordinates to domain
7    float fx = (float)wWidth / (float)N;
8    float fy = (float)wHeight / (float)N;
9
10   int xPos = (int) ((float) TouchPos.x() / ((float) fx)) + 1 ;
11   int yPos = (int) ((float) TouchPos.y() / ((float) fy)) + 1 ;
12
13   //add density
14   if (p0.state() && Qt::TouchPointPressed && value==0)
15   {
16     solver.addDensity(xPos, yPos, density_strength, brushDensity,
         ColorLayer < QUANTLAYER ? ColorLayer : 0);
17     destSize = ci::Vec2f((float)xPos,(float)yPos);
18   }
19
20   //add force
21   if (p0.state() && Qt::TouchPointPressed && value==1 && !comb)
22   {
23     float xforce = TouchVel.x() != 0 ? (((float)PrecitionVel / (float) abs(
         TouchVel.x())) + 1.0f) : 1.0f;
24     float yforce = TouchVel.y() != 0 ? (((float)PrecitionVel / (float) abs(
         TouchVel.y())) + 1.0f) : 1.0f;
25     solver.addVelocity(xPos, yPos, ((float) TouchVel.x()) / xforce, ((float
         )TouchVel.y()) / yforce, brushVelocity);
26   }
27
28   lastx = p0.pos().x();
29   lasty = p0.pos().y();
30   TouchLastPos=p0.pos().toPoint();
31 }
```

4.6.2 Movimento de Pente

O movimento do pente é realizado para criar desenhos como os chamados de *Nonpareil* (Figura 17-(a)) e *Gel Git* (Figura 17-(b)) no marmoreio em papel. Para isto, os pentes utilizados são de diferentes quantidades de dentes; porém nosso protótipo utiliza como base um pente padrão que tem aproximadamente 120 dentes (Figura 2-(a) do Capítulo 1) com o qual poderá ser obtido o padrão *Nonpareil*.

Figura 17 – Padrões obtidos com o pente: (a) Padrão *Nonpareil*, (b) Padrão *Gel Git*.

(a) (b)

Fonte – (34)

Na Listagem 26 é apresentado o trecho de código para aplicar o movimento do pente. A mesa digitalizadora usada por nosso protótipo reconhece um toque, portanto transformamos esse toque em vários toques ao mesmo tempo. Para exemplificar, vamos a apresentar nesta listagem o caso de reconhecimento de dois toques numa tela multi-ponto, aonde o método `applyPoints()` realiza as múltiplas espalhamentos. Nas Linhas 13-20 é feito o cálculo do número de dentes do pente. As Linhas 22-23 realizam o cálculo da intensidade do movimento do pente e nas Linhas 26-49 é configurado a direção do movimento do pente (direita, esquerda, acima, embaixo) para cada dente.

Listagem 26 – OenGLWdget.cpp: método `applyPoints()`.

```
1  void OpenGLWidget::applyPoints(QTouchEvent::TouchPoint &p0, QTouchEvent::
       TouchPoint &p1, int value)
2  {
3    QPoint TouchPos1 = p1.pos().toPoint();
4    QPoint TouchVel1 = TouchPos1 - TouchLastPos;
5
6    // Convert motion coordinates to domain
7    int xPos1 = (int) ((float) TouchPos1.x() / ((float) fx)) + 1 ;
8    int yPos1 = (int) ((float) TouchPos1.y() / ((float) fy)) + 1 ;
9
10   //add velocities (comb)
11   if (p1.state()  && Qt::TouchPointPressed && value==1 && comb)
12   {
13     int yPosN[120];int xPosN[120];
14     int spaceH = wHeight/120;
```

```
15      int spaceW = wWidth/120;
16
17      //Points on y axis
18      for(int i=0; i<120; i++) yPosN[i] = i*spaceH;
19      //Points on x axis
20      for(int i=0; i<120; i++)  xPosN[i] = i*spaceW;
21
22      float xforce = TouchVel1.x() != 0 ? (((float)PrecitionVel / (float) abs
            (TouchVel1.x())) + 1.0f) : 1.0f;
23      float yforce = TouchVel1.y() != 0 ? (((float)PrecitionVel / (float) abs
            (TouchVel1.y())) + 1.0f) : 1.0f;
24
25      //(0:right, 1:left, 2:top, 3:bottom)
26      switch(pattern)
27      {
28      case 0:
29      for(int i=0; i<120; i++)
30        solver.addVelocity(xPos,yPosN[i],((float)TouchVel1.x())/xforce,0,
              brushVelocity);
31      break;
32      case 1:
33      for(int i=0; i<120; i++)
34        solver.addVelocity(xPos,yPosN[i],-((float) TouchVel1.x())/xforce,0,
              brushVelocity);
35      break;
36      case 2:
37      for(int i=0; i<120; i++)
38        solver.addVelocity(xPosN[i],yPos,0,-((float)TouchVel1.y())/yforce,
              brushVelocity);
39      break;
40      case 3:
41      for(int i=0; i<120; i++)
42        solver.addVelocity(xPosN[i],yPos,0,((float)TouchVel1.y())/yforce,
              brushVelocity);
43      break;
44      default:
45      for(int i=0; i<120; i++)
46        solver.addVelocity(xPos,yPos,((float)TouchVel1.x())/xforce, ((float)
              TouchVel1.y())/yforce, brushVelocity);
47      break;
48      }
49    }
50
51    lastx = p1.pos().x();
52    lasty = p1.pos().y();
53    TouchLastPos=p1.pos().toPoint();
```

4.7 Condições Finais

Neste capítulo foi apresentado o desenvolvimento da técnica para criação das tintas de marmoreio digital, em que foi usado o modelo *syfluid* de (30) para realizar o escoamento do fluido. A nossa contribuição, em relação aos outros sistemas de marmoreio digital, está dado pelo desenvolvimento da técnica chamada de *3D point light* para realizar os efeitos de luz sobre as tintas, fornecendo uma aparência mais realista; e a integração dos blocos apresentados na Figura 11, a fim de transformá-los em tintas de marmoreio digital.

5 Resultados e Discussões

5.1 Condições Iniciais

Neste capítulo vamos apresentar o uso do protótipo descrevendo as principais funcionalidades da interface natural desenvolvida, mostrando alguns padrões que foram obtidos por nosso protótipo. Discutiremos alguns resultados parciais obtidos no decorrer da pesquisa, pois eles foram importantes na escolha da metodologia apresentada neste trabalho. Finalizamos com recomendação de alguns trabalhos que poderiam ser feitos baseados nesta pesquisa.

5.2 Uso do Protótipo

Para ilustrar a utilização do nosso protótipo durante a criação de padrões de marmoreio digital, vamos apresentar as principais interfaces de nosso sistema de marmoreio digital.

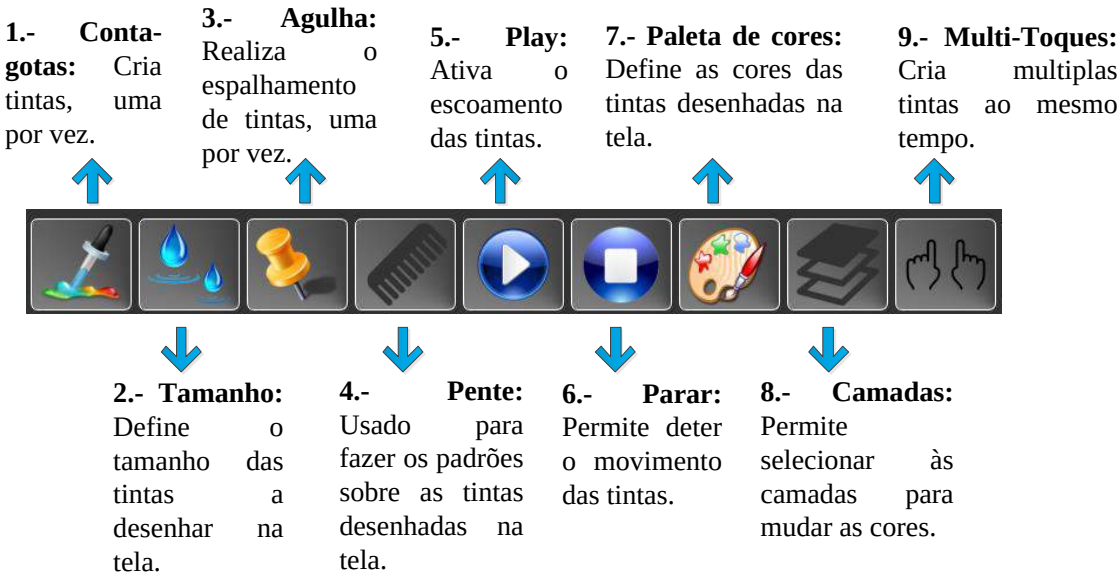
A primeira interface é apresentada na Figura 18, constando de três áreas:

- A região de desenho: onde serão criados os padrões de marmoreio digital;
- A barra de configurações: que contém as diferentes ferramentas usadas pelos usuários, cores, tamanhos das tintas e a seleção das camadas;
- A barra padrão: que apresenta as opções padrão do protótipo, tais como salvar, sair e novo.

Figura 18 – Componentes da interface principal.

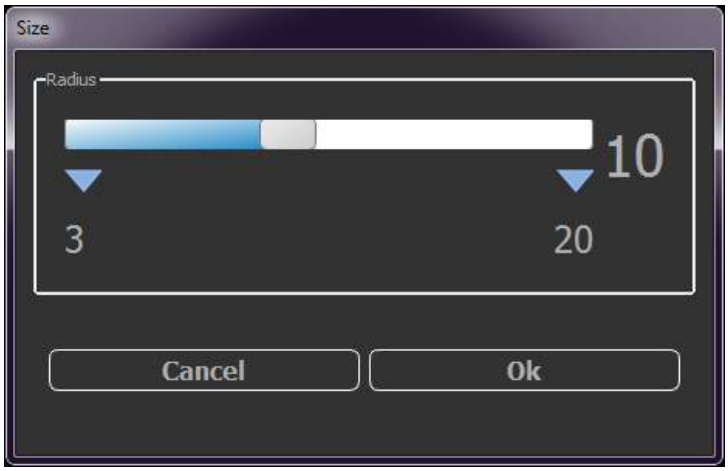
As diversas ferramentas consideradas para nosso marmoreio digital estão apresentadas ao lado superior da região de desenho. A Figura 19 descreve brevemente as funções de cada opção.

Figura 19 – Barra de configurações: Descrição dos nove botões do lado esquerdo da interface principal.



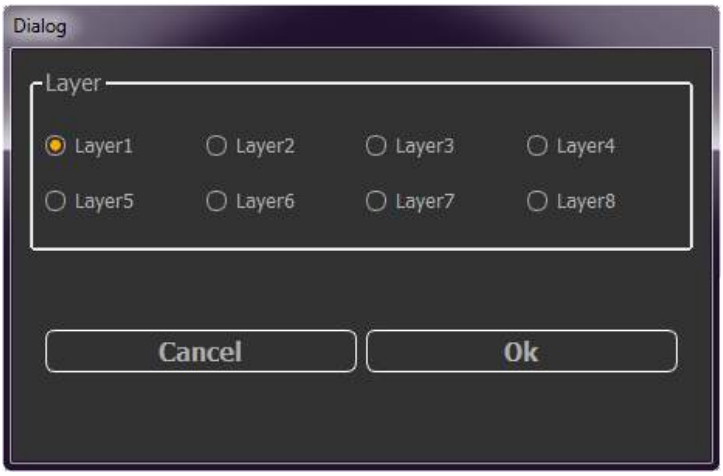
A segunda interface a mencionar é aquela que controla o tamanho das tintas digitais (Figura 20). Esta interface é invocada mediante o botão  da barra de configurações. Está definido pelo tamanho (raio) da tinta a criar, e pode variar entre os valores de 3 e 20 que são fatores para obter tintas entre 7.5mm e 50mm, que é o tamanho real numa grade 256×256 com resolução 1920×1080.

Figura 20 – *Size radius*: Configuração do tamanho das texturas.



A terceira interface a destacar é aquela usada para mudar uma cor em uma camada existente, ou seja se o usuário deseja mudar a cor de uma tinta inserida anteriormente, pode realizar esta mudança mediante o botão  da barra de configurações. Em nosso protótipo foram implementadas oito camadas, cada uma apresentada mediante o termo *LayerN* (Figura 21), onde *N* representa o número da camada que se quer mudar a cor.

Figura 21 – *Change Color Layer*: Configuração de uma nova cor numa camada existente.



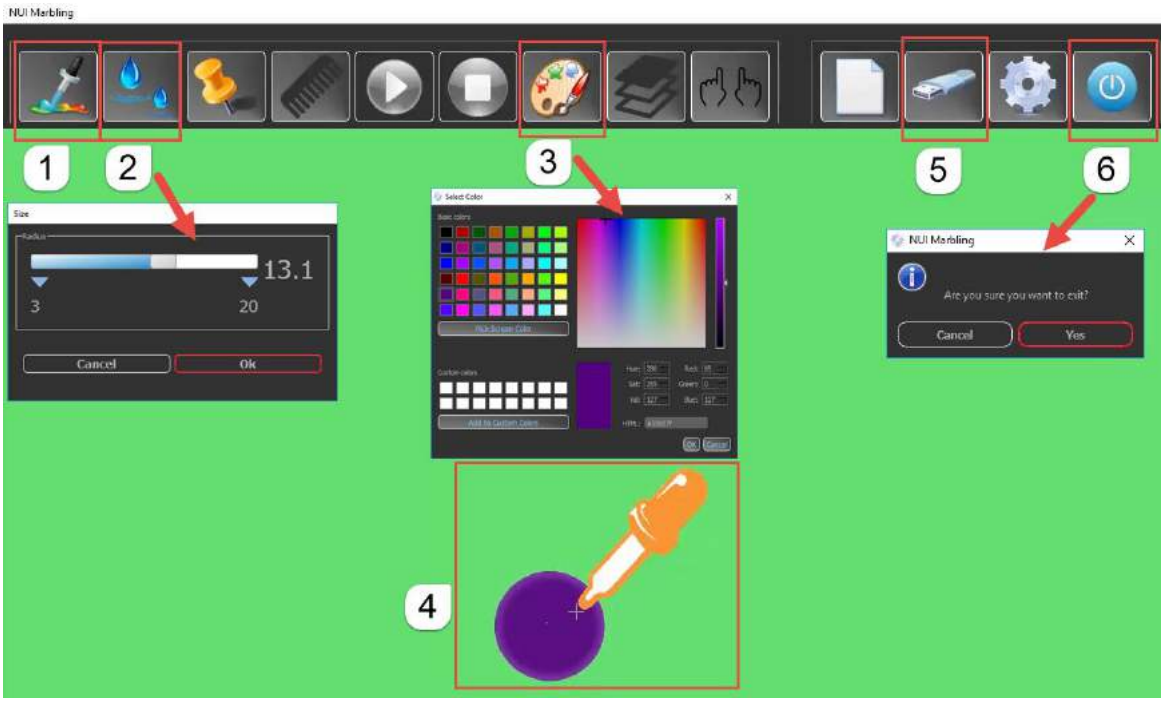
5.2.1 Criação de tintas

Com as informações anteriormente descritas, vamos criar as primeiras tintas em nosso protótipo. Esta tarefa tem dois processos, o primeiro criando uma tintas por vez e o segundo criando múltiplas tintas, ambos os processos com a ferramenta *Conta-gotas* (Figura 22). estes processos são diferenciados pela seleção do tipo de entrada na caneta digital.

Na Figura 22 temos os passos numerados desde 1 até 6 para criar uma tinta, em que o usuário deverá:

1. Escolher a ferramenta *conta-gotas* da barra de configurações;
2. Selecionar a cor da tinta;
3. Escolher o tamanho da tinta a desenhar: Este passo é opcional devido a que o tamanho padrão das tintas está dado pelo valor 10;
4. Criar a tinta na região de desenho: Dando um toque sobre a mesa digitalizadora;
5. Salvar seu desenho: Quando estiver finalizado o desenho, o usuário tem a opção de salvá-lo no computador como um arquivo PNG;
6. Sair do protótipo.

Figura 22 – Criação de tintas digitais: Criar tintas com conta-gotas.



5.2.2 Espalhamentos

Nosso protótipo de marmoreio digital pode realizar dois tipos de espalhamentos sobre as tintas, cada tipo com uma ferramenta diferente:

- Ferramenta Pente: Permite realizar os espalhamentos sobres as tintas, a fim de simular as ondulações aplicadas em mais de um ponto sobre a região de desenho (Figura 23);
- Ferramenta Agulha: Permite realizar espalhamentos sobre as tintas a fim de simular as ondulações aplicadas em um ponto sobre a região de desenho (Figura 24).

Figura 23 – Uso da ferramenta pente: Criação de um padrão de marmoreio digital com três toques em sentido ascendente.

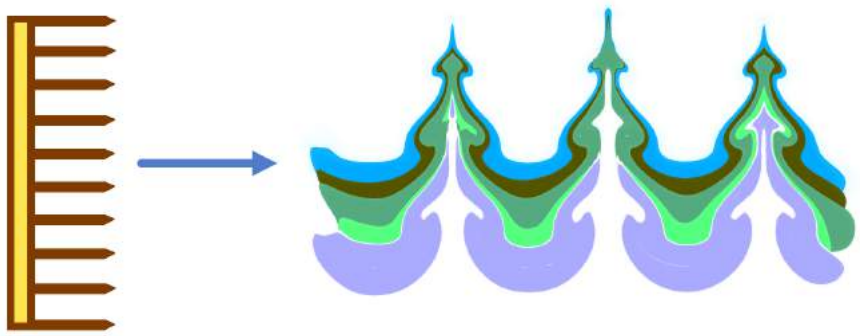
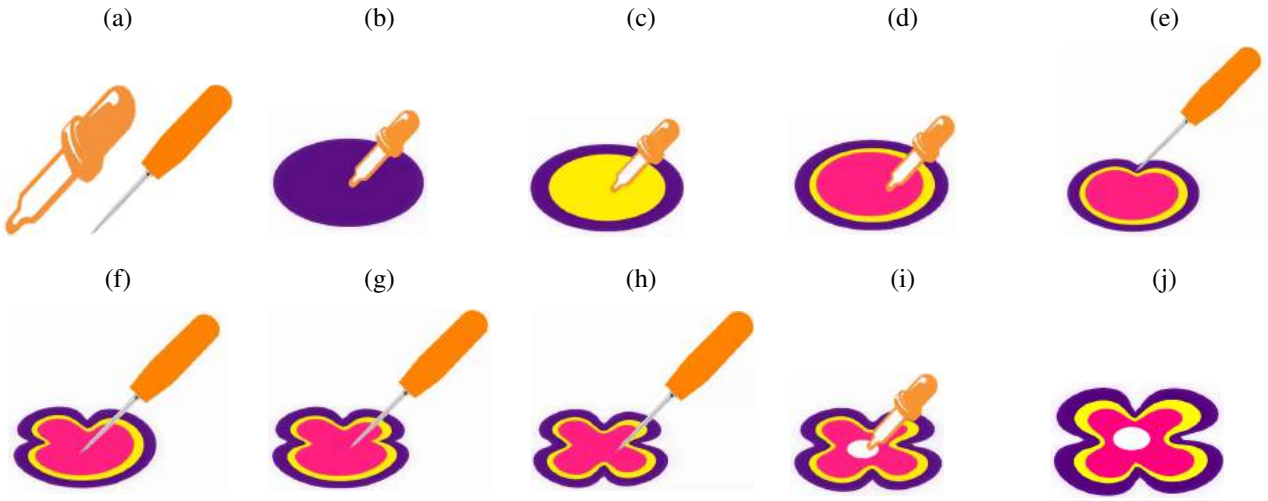


Figura 24 – Uso da ferramenta agulha: (a) Ferramentas conta-gotas e agulha, (b) Primeira camada, (c) Segunda camada, (d) Terceira camada, (e) Movimento descendente (f) Movimento esquerda-direita, (f) Movimento direita-esquerda, (e) Movimento ascendente, (g) Quarta camada, e (h) Padrão final.



5.2.3 Mudar Cores de Tintas

Nosso protótipo fornece mudanças nas cores das tintas existentes, ou seja, no caso em que um usuário deseja mudar a cor das tintas criadas na região de desenho, ele pode realizar esta tarefa. Na Figura 25-(a) podemos observar até 5 cores distintas de tintas na região de desenho. Vamos a realizar a mudança nas cores das tintas amarelo por tintas na cor marrom. Para isto, considerarmos enumerar os passos para realizar esta tarefa, a seguir:

1. Criar as tintas na região de desenho;
2. Escolher a opção camadas da barra de configurações;
3. Selecionar a camada da cor a mudar na janela emergente (*Layer3* na Figura 25-(a));
4. Selecionar a cor com que se quer substituir.

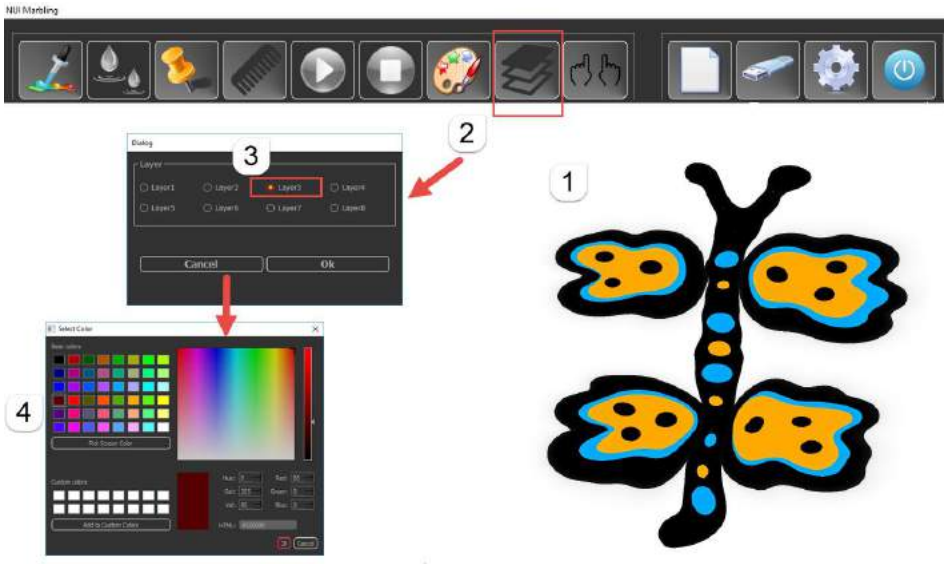
Os resultados da mudança das tintas são apresentados na Figura 25-(b).

5.3 Resultados Visuais

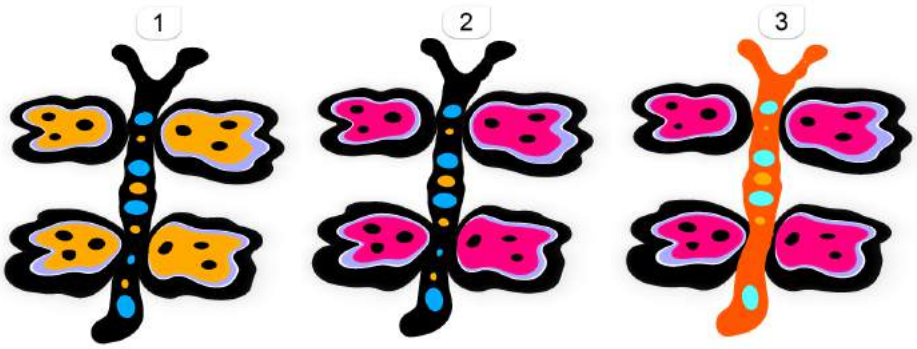
Vamos apresentar os resultados obtidos realizando os espalhamentos das tintas com a ferramenta agulha. As Figuras 26-(a) e 26-(c) representam os padrões que foram feitos com nosso protótipo. As Figuras 26-(b) e 26-(d) representam os padrões que foram feitos com a técnica de marmoreio em papel, tomados como referência.

Figura 25 – Mudar as cores das tintas: (a) Padrão inicial, (b) Três mudanças do padrão inicial.

(a)



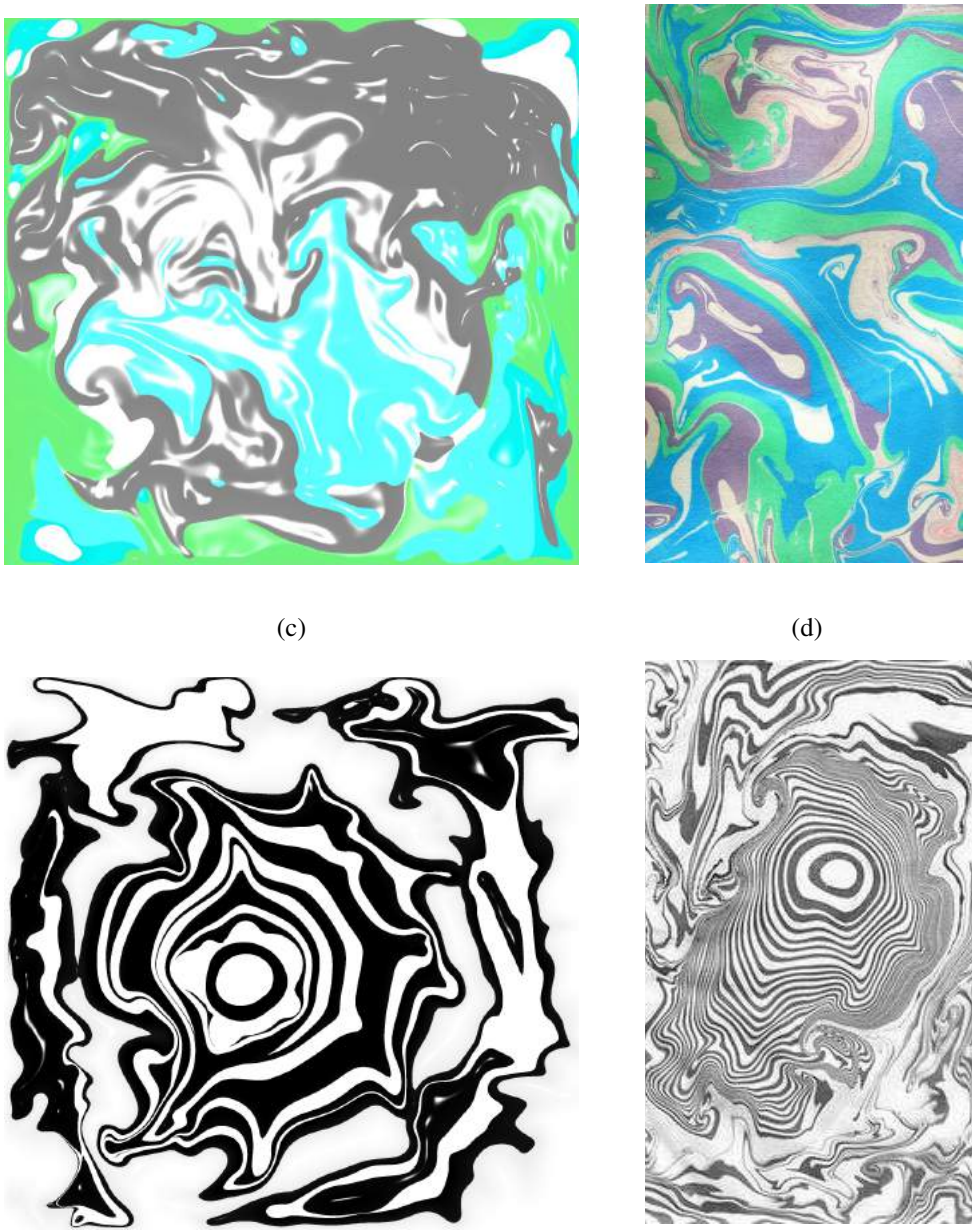
(b)



Para apresentar a utilidade da ferramenta pente na interação com as tintas, vamos apresentar os resultados obtidos mediante o chamado de padrão *Nonpareil* do marmoreio em papel. A Figura 27-(a) representa o desenho que foi feito com nosso protótipo com uma passada do pente. A Figura 27-(b) com duas passadas da ferramenta pente na mesma direção (da direita para a esquerda). A Figura 27-(c) representa o padrão feito com a ferramenta pente aplicado na direção de cima para baixo, três vezes.

Figura 26 – Criação de padrões com a ferramenta agulha. (a)-(c) Resultados em nosso protótipo, (b)-(d) Modelo de referência no marmoreio em papel.

(a) (b)

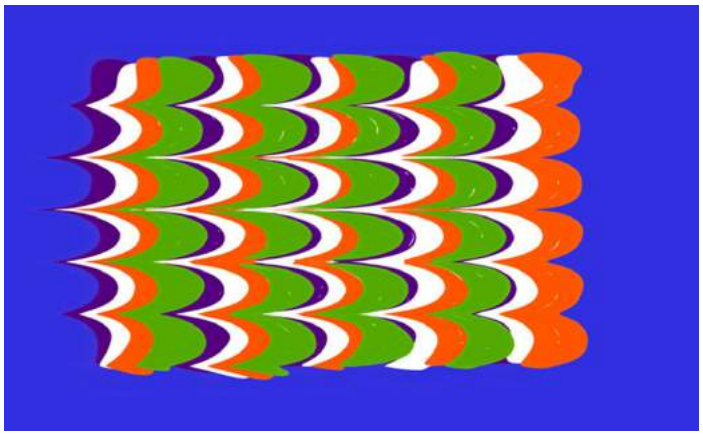


(c) (d)

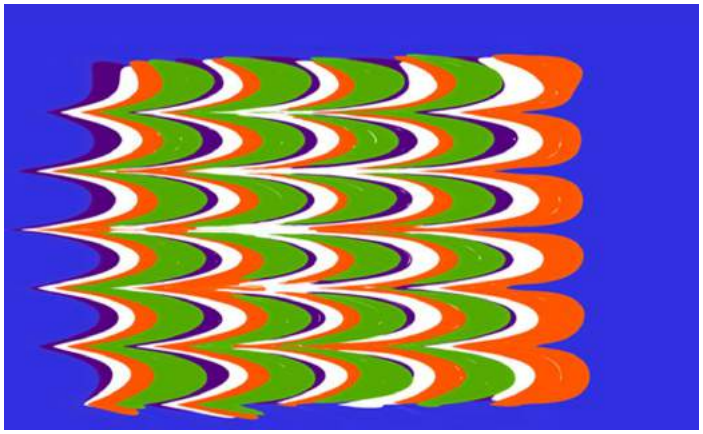
Fonte – (b) - (35), (c)-(36)

Figura 27 – Criação de padrões *Nonpareil* com a ferramenta pente. (a) Uma passada, da direita para a esquerda; (b) Duas passadas, da direita para a esquerda; (c) Três passadas de cima para baixo.

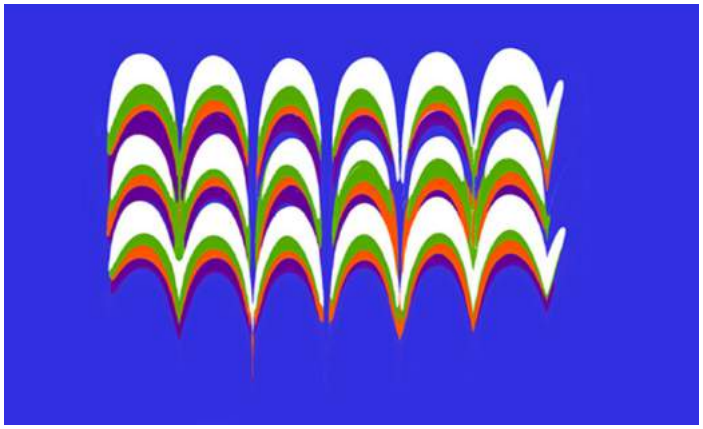
(a)



(b)

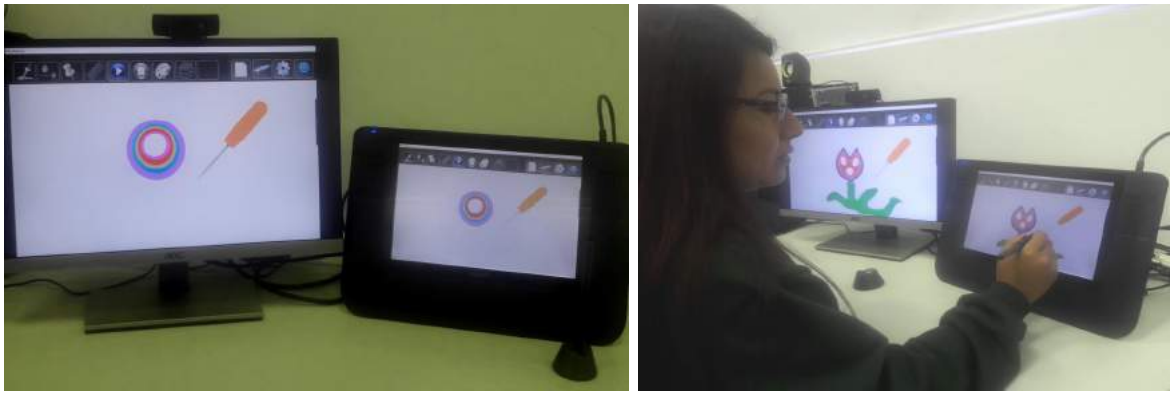


(c)



A seguir, apresentamos na Figura 28 a interação de um usuário com o nosso protótipo, através da mesa digitalizadora.

Figura 28 – Interação de usuário: (a) Equipes tecnológicos usados por nosso protótipo; (b) Uso da caneta digital sobre a mesa digitalizadora para criação de padrões de marmoreio digital em nosso protótipo.



(a) (b)

5.4 Resultados Numéricos

Os testes para o desempenho foram realizados em dois computadores. Os ambientes de cada um são descritos nas Tabelas 4 e 5. No primeiro ambiente temos o processador que denominaremos GPU1 e o segundo ambiente terá os processadores a chamar de CPU e GPU2.

Tabela 4 – Ambiente do processador GPU1.

Recurso	**Característica**	**Descrição**
S.O.	Edição	Windows 7 Ultimate SP1
	Tipo	64 bits
Processador	Cache	3MB
	Tecnologia	Dual Core
Memória	Memória instalada	4GB
	Tipo / Velocidade	DDR3 / 1333MHz
Monitor	Tamanho	24.0"
	Resolução	1920 x 1080
	Tecnologia - Iluminação	LED
	Tecnologia - Display	Full HD, Multi-toques
Vídeo	Processador gráfico	NVIDIA GeForce GT 540M
	RAM	1GB dedicated
	CUDA cores	96
Mesa Digitalizadora	Tipo de Entrada	Mouse & Caneta
	Área Ativa	12.8"(325.1 mm) x 8"(203.2 mm)
	Modelo	Wacom PTK840

Como dito no capítulo 4, o desenvolvimento de nosso protótipo foi realizado mediante

Tabela 5 – Ambiente dos processadores CPU e GPU2.

Recurso	**Característica**	**Descrição**
S.O.	Edição	Windows 8.1 Pro
	Tipo	64 bits
Processador	Cache	10MB
	Tecnologia	Core i7
Memória	Memória instalada	32GB
	Tipo / Velocidade	DDR3 / 1333MHz
Monitor	Tamanho	21.5"
	Resolução	1920 x 1080
	Tecnologia - Iluminação	LED
	Tecnologia - Display	Full HD, Multi-toques
Vídeo	Processador gráfico	NVIDIA GeForce GTX 970
	RAM	4GB dedicated
	CUDA cores	1664
Mesa Digitalizadora	Tipo de Entrada	Mouse & Caneta
	Área Ativa	12.8"(325.1 mm) x 8"(203.2 mm)
	Modelo	Wacom PTK840

a linguagem de programação C++, realizando a simulação do escoamento de fluidos mediante programas *shaders* usando a linguagem GLSL a fim de obter resultados em tempo real. Cabe ressaltar que, nos testes foram utilizadas 20 camadas, cada camada com 20 tintas de tamanho 20 (o maior tamanho), as quais ao finalizar a criação da última camada foram espalhadas todas as tintas digitais com nossa ferramenta pente.

A Tabela 6 apresenta os resultados obtidos em *frames* por segundo (FPS) em cada domínio configurado para os três processadores (CPU, GPU1, GPU2) com as características mencionadas nas Tabelas 4 e 5.

Tabela 6 – Resolução do Grade vs. Processador. FPS obtidos.

	CPU	**GPU1**	**GPU2**
256×256	2.85	10	41.6
512×512	0.35	1.42	27
1024×1024	0.05	0.43	6.36

A comparação dos FPS obtidos por nosso protótipo em relação ao protótipo de Jin *et al.* (5) chamado de *Realtime Marbling* é apresentado na Tabela 7.

Além de avaliar o desempenho em relação ao tipo de processador, realizamos uma série de simulações usando nosso protótipo, comparando o desempenho em relação ao efeito visual

Tabela 7 – Resolução do Grade vs. Sistema . FPS obtidos.

	Realtime Marbling	**Nosso Protótipo**
256×256	37.6	41.6
512×512	15.0	27
1024×1024	4.2	6.36

desenvolvido tal como o filtro *shock filter* (Seção 4.5.1). Estes resultados são apresentados na Tabela 8.

Tabela 8 – Resolução do Grade vs. Efeito Visual. FPS obtidos.

	Sem *Shock Filter*	**Com *Shock Filter***
256×256	43.4	41.6
512×512	29.5	27
1024×1024	7.4	6.36

Estes últimos resultados garantem que o uso de um solucionador para transformar as transições suaves que causam os borramentos dos contornos do *solver*, como o implementado neste trabalho, não afeta a precisão do modelo físico.

5.5 Limitações

Nosso protótipo foi desenvolvido para enviar um *feedback* em tempo real sempre que tenha um processador gráfico dedicado de qualquer fabricante sob o Sistema Operacional Windows. Em computadores com apenas processador central não fornecemos uma experiência natural ao usuário.

Apesar do fato de podermos criar tintas digitais com múltiplas cores, seu numero total não pode ultrapassar 20 cores. Esta quantidade máxima de cores possível depende da quantidade de RAM que a placa gráfica possui. Uma fórmula aproximada que usamos para calcular a RAM de vídeo necessária por cor se da por meio de:

$$((2 * GridSize * GridSize) + QuantityColorGradient) * 4Bytes \tag{1}$$

Onde, *GridSize* é a discretização do domínio e $QuantityColorGradient$ é a quantidade de tons a criar em cada gradiente (cor). Assim por exemplo, nosso protótipo numa grade de 256×256 temos: $((2 * 256 * 256) + 2048) * 4 = 532480$ o que significa aproximadamente 532kB do uso de RAM de vídeo por cor (camada). Além disso, deve-se considerar o desempenho e os outros processos que se executa ao mesmo tempo na placa gráfica.

5.6 Discussões

5.6.1 Uso do processamento na GPU

No inicio da implementação deste trabalho se utilizou o próprio processador da CPU para realizar la simulação do escoamento dos fluidos, que é a base para realizar a criação de tintas de marmoreio digital, com a finalidade de realizar uma projeção do uso necessário da GPU. Esta projeção foi feita sobre o processamento da solução linear para simulação do escoamento de fluidos (método `linsolve()`, Seção 4.3.1). A projeção de recursos foi feita sobre duas GPU e uma CPU com as seguintes características:

- CPU: Processador Intel Xeon E5-2620 v2, 15M Cache, 2.10 GHz.
- GPU1: NVIDIA GeForce GTX 750 com 512 núcleos.
- GPU2: NVIDIA Tesla K20 com 2496 núcleos.

O processamento da solução linear foi avaliado em relação ao tempo obtido para 1, 10 e 100 frames. Os resultados obtidos são mostrados na Tabela 9

Tabela 9 – Frame vs. Processador. Tempos obtidos em segundos.

	1 frame	**10 frames**	**100 frames**
CPU	1.19	22.79	233
GPU1	1.597	2.512	54.05
GPU2	0.85	0.87	1.8

A Tabela 10 apresenta o *speedup* atingido para 100 frames (necessários para uma resolução *FullHD*). Assim, o cálculo para a GPU1 foi obtido: $233/54.05 = 4,3\times$, em que usamos $\times$ para representar o *speedup*. O cálculo para a GPU2 foi obtido: $233/1.8 = 129,4\times$.

Tabela 10 – *Speedup* obtido por cada processador. Tempo em segundos.

	Tempo	**Speedup**
CPU	233	-
GPU1	54.05	4,3×
GPU2	1.8	129×

Segundo Weerd (37) a percepção humana consegue processar com realismo ambientes escuros ou filmes sem som acima dos 24 FPS. A maioria das pessoas não percebem videos muito mais suaves acima de 60 FPS. Com esta informação foi definido como meta atingir 30 FPS em resolução FullHD. Assim o cálculo do FPS para a CPU é obtido: $100/233 = 0.42$ FPS,

Tabela 11 – FPS obtidos por cada processador. Tempo em segundos.

	Tempo	**FPS**
CPU	233	0.42
GPU1	54.05	1.85
GPU2	1.8	55.5

para a GPU1: $100/54.05 = 1.85$ FPS e para a GPU2: $100/1.8 = 55.5$ FPS. Estes resultados são mostrados na Tabela 11.

Baseados nas informações da Tabela 11 a necessidade de implementar nosso protótipo usando alguma linguagem de programação paralela foi justificada para uma simulação em tempo real.

5.6.2 Valores de Precisão na Nitidez das Tintas

Para ter uma nitidez das tintas aceitável foram realizadas uma série de simulações para obter o valor de precisão mais adequado com nosso objetivo. Os valores que estão diretamente relacionados com este objetivo específico são: `treshold` e `shockMagnitude` do arquivo fdraw.glsl descrito na Secção 4.5.1.

A Figura 29 apresenta os resultados obtidos em cada par de combinação `treshold`-`shockMagnitude`. Baseados nestes resultados, resolvemos estabelecer o par de combinação da Figura 29-(l) como o melhor valor de precisão para nitidez das tintas criadas por nosso protótipo.

5.7 Considerações Finais

Os resultados apresentados neste capítulo demonstram que nosso protótipo foi desenvolvido para se adaptar tanto nos dispositivos eletrônicos de entrada convencionais (mouse) quanto aqueles de reconhecimento múltiplo (tela de toques multi-ponto, mesa digitalizadora, dentre outras) fornecendo assim, maior e fácil exploração ao usuário com o nosso protótipo de marmoreio digital. Além disso, foi providenciado que a interação é feita em tempo real, que é uma necessidade aprimorante neste tipo de aplicativos.

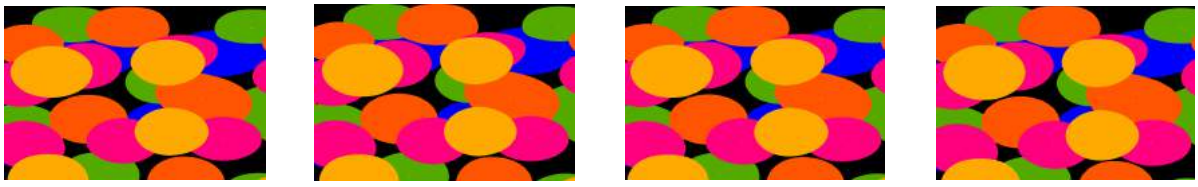
Figura 29 – Resultados parciais para obter a nitidez das tintas: Cada par representa o valor de `treshold-shockMagnitude`.

(a) 0.05-0.05 (b) 0.05-0.9 (c) 0.05-2.0 (d) 0.05-6.0

(e) 0.1-0.05 (f) 0.1-0.9 (g) 0.1-2.0 (h) 0.1-6.0

(i) 0.5-0.05 (j) 0.5-0.9 (k) 0.5-2.0 (l) 0.5-6.0

(m) 0.8-0.05 (n) 0.8-0.9 (o) 0.8-2.0 (p) 0.8-6.0



(q) 0.9-0.05 (r) 0.9-0.9 (s) 0.9-2.0 (t) 0.9-6.0

6 Conclusões e Trabalhos Futuros

O presente capítulo apresenta as conclusões alcançadas no desenvolvimento da pesquisa, as principais contribuições e os trabalhos que poderão ser encaminhados em base desta pesquisa.

6.1 Conclusões

Este trabalho foi inspirado pelas necessidades descritas na Seção 1.1, em que apesar do fato de que existem sistemas de marmoreio digital, nenhum deles fornece uma interface natural para interação com os usuários. Este objetivo foi atingido mediante a implementação de uma interface natural utilizando uma tela multi-toques como superfície para criação de padrões de marmoreio, fornecendo assim uma interação de fácil aprendizagem em que todas as ferramentas a utilizar pelos usuários são apresentadas numa única interface.

Para realizar a simulação das tintas de marmoreio digital, realizamos a adaptação da abordagem da dinâmica dos fluidos computacionais apresentada por Stam. Para implementar as cores e espalhamentos das tintas, que são processos próprios do marmoreio, utilizamos a abordagem de Jin *et. al*. Além disso melhorarmos a qualidade das tintas, pois elas sofrem de borramentos devido às interações para resolver as equações lineais, fornecendo a nitidez das imagens utilizando o filtro chamado de *Shock Filter* e a técnica *Multisample anti-aliasing*.

As ferramentas principais utilizadas no desenvolvimento do protótipo foram o Qt *framework* e a biblioteca OpenGL e que como definimos no capítulo 3, as tintas foram tratadas computacionalmente como texturas, as quais foram renderizadas usando a GPU em que os programas *shader* da biblioteca Cinder utilizaram GLSL a fim de fornecer aos usuários um *feedback* em tempo real, melhorando assim o desempenho de nosso protótipo em relação de uma implementação realizada numa CPU apenas.

Duas diretrizes fornecidas pela HCI foram consideradas no desenvolvimento da interface natural. A primeira foi que o usuário pudesse sentir que a interface seria uma extensão do seu corpo, pois ele utiliza seus dedos para interagir com nosso protótipo. A segunda foi considerar que todas as ferramentas foram baseadas nas ferramentas reais utilizadas no marmoreio em papel, tais como o pente, a escova e o conta-gotas, ou seja, temos considerado o contexto real no desenvolvimento de nosso protótipo.

6.2 Contribuições

O trabalho de Jin *et al.* (5) entre outros trabalhos de marmoreio digital estudados no capítulo 2 forneceram os conhecimentos gerais para realizar o desenvolvimento de nosso protó-

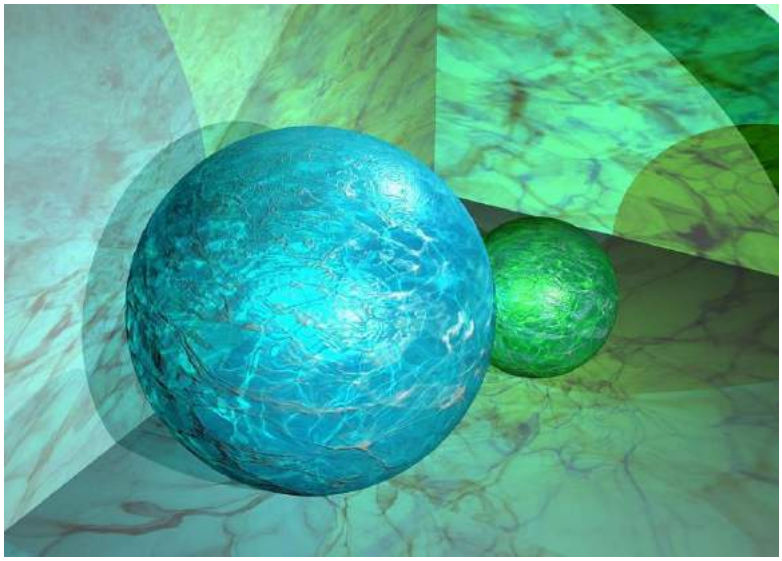
tipo. Eles mencionam as diversas técnicas a utilizar para realizar a implementação de marmoreio digital, contudo apresentamos contribuições em alguns aspectos:

- O protótipo foi desenvolvido na linguagem GLSL permitindo executar em qualquer tipo de processador gráfico que suporta a biblioteca OpenGL a partir da versão 2.0;
- Fornecemos uma interface mais natural em relação a todos os sistemas de marmoreio estudados;
- A vantagem interativa de ter apenas uma interface com todas as opções de marmoreio digital, além de executá-los em tempo real;
- Este trabalho pode ser base para outros na simulação do escoamento de fluidos devido a que fornecemos um mapeamento detalhado da arquitetura de nosso protótipo e as listagens apresentadas no capítulo 3.

6.3 Trabalhos Futuros

O marmoreio na atualidade tem aplicações no dia a dia, tais como: Arquitetura, desenho de interiores, beleza, indústria automobilística, entre outras. Estas aplicações poderiam levar a uma aplicação da presente pesquisa a objetos 3D, renderizando o padrão obtido em malhas geométricas para tal fim (Figura 30).

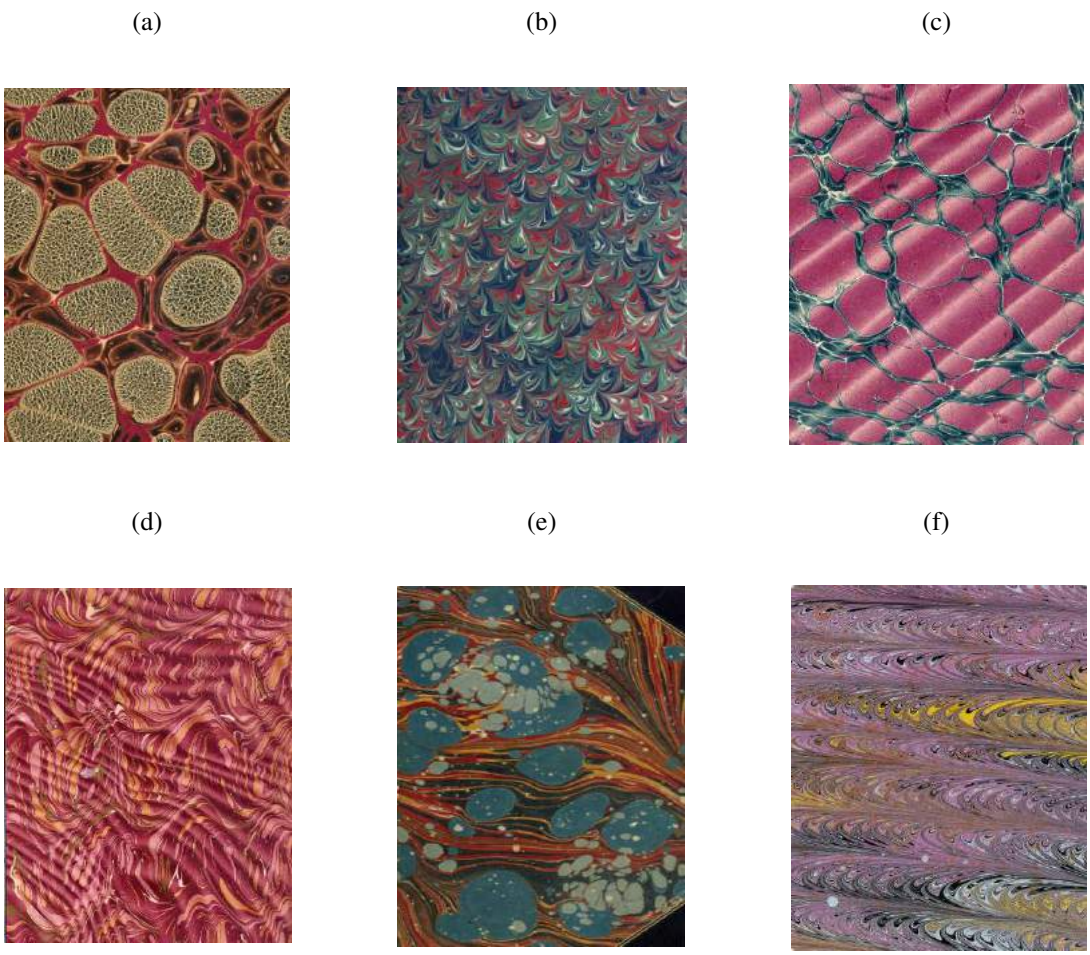
Figura 30 – Marmoreio digital numa malha 3D: Esferas geométricas renderizadas com texturas em baseadas no marmoreio.



Fonte – (38).

Com a finalidade de fornecer aos usuários maiores opções de criação de padrões poderia se implementar outros espalhamentos sobre as tintas, baseados na implementação realizada neste trabalho. A biblioteca da Universidade de Washington (39) fornece 32 distintos padrões desenhados no mundo, através do marmoreio em papel. Na Figura 31 podemos observar alguns dos padrões obtidos no marmoreio em papel.

Figura 31 – Alguns padrões de marmoreio em papel: (a) *Romantic*, (b) *Serpentine*, (c) *Spanish*, (d) *Spanish moiré*, (e) *Dahlia*, (f) *Double Marble*.



Fonte – (39)

Referências

1 MAO, X.; SUZUKI, T.; IMAMIYA, A. Atelierm: A physically based interactive system for creating traditional marbling textures. In: *Proceedings of the 1st International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*. New York, NY, USA: ACM, 2003. (GRAPHITE '03), p. 79–ff. ISBN 1-58113-578-5. Disponível em: <http://doi.acm.org/10.1145/604471.604489>. Citado 4 vezes nas páginas 1, 2, 4 e 8.

2 MAURER-MATHISON, D. V. *The Ultimate Marbling Handbook: A Guide to Basic and Advanced Techniques for Marbling Paper and Fabric*. 1st. ed. [S.l.]: Watson-Guptill Publications, 1999. Citado na página 1.

3 COMMONS, C. *Cómo pintar en agua como en papel*. 2014. <https://www.youtube.com/watch?v=NqedBekgLdo>. [Online; accessed 2016-04-25]. Citado na página 1.

4 UNDER, M. *A Morning of Paper Marbling*. 2013. <https://parkslibrarypreservation.wordpress.com/2013/08/13/a-morning-of-paper-marbling/>. [Online; accessed 2016-04-12]. Citado na página 2.

5 JIN, X.; CHEN, S.; MAO, X. Computer-generated marbling textures: A gpu-based design system. *Computer Graphics and Applications, IEEE*, v. 27, n. 2, p. 78–84, March 2007. ISSN 0272-1716. Citado 10 vezes nas páginas 2, 4, 5, 8, 9, 10, 23, 30, 68 e 73.

6 LU, S. et al. Mathematical marbling. *Computer Graphics and Applications, IEEE*, v. 32, n. 6, p. 26–35, Nov 2012. ISSN 0272-1716. Citado 6 vezes nas páginas 2, 4, 7, 9, 11 e 23.

7 CORPORATION., N. *What is GPU accelerated computing?* 2016. <http://www.nvidia.com/object/what-is-gpu-computing.html>. [Online; accessed 2016-02-26]. Citado 2 vezes nas páginas 2 e 3.

8 ZHAO, H. et al. Atelierm++: A fast and accurate marbling system. *Multimedia Tools Appl.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 44, n. 2, p. 187–203, set. 2009. ISSN 1380-7501. Disponível em: <http://dx.doi.org/10.1007/s11042-009-0290-z>. Citado 3 vezes nas páginas 3, 8 e 9.

9 LU, S. et al. Real-time image marbleization. *Multimedia Tools Appl.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 64, n. 3, p. 795–808, jun. 2013. ISSN 1380-7501. Disponível em: <http://dx.doi.org/10.1007/s11042-012-0989-0>. Citado 3 vezes nas páginas 4, 9 e 10.

10 CRANOR, L. F.; GARFINKEL, S. (Ed.). *Security and usability : designing secure systems that people can use*. Beijing, Sebastapol, CA: O'Reilly, 2005. ISBN 978-0-596-00827-7. Disponível em: <http://opac.inria.fr/record=b1133414>. Citado na página 4.

11 WIGDOR, D.; WIXON, D. *Brave NUI World: Designing Natural User Interfaces for Touch and Gesture*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 0123822319, 9780123822314. Citado na página 5.

12 STAM, J. Stable fluids. In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999. (SIGGRAPH '99), p. 121–128. ISBN 0-201-48560-5. Disponível em:

<http://dx.doi.org/10.1145/311535.311548>. Citado 7 vezes nas páginas 5, 7, 15, 21, 22, 35 e 38.

13 SHACKEL, B.; RICHARDSON, S. J. *Human Factors for Informatics Usability*. 1st. ed. New York, NY, USA: Cambridge University Press, 2008. ISBN 0521067308, 9780521067300. Citado na página 5.

14 STAM, J. Real-time fluid dynamics for games. In: *Proceedings of the Game Developer Conference*. Citeseer, 2003. v. 18. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.12.6736>. Citado 2 vezes nas páginas 7 e 19.

15 HARRIS, M. Fast fluid dynamics simulation on the gpu. In: *ACM SIGGRAPH 2005 Courses*. New York, NY, USA: ACM, 2005. (SIGGRAPH '05). Disponível em: <http://doi.acm.org/10.1145/1198555.1198790>. Citado 8 vezes nas páginas 7, 15, 18, 19, 20, 21, 22 e 38.

16 ACAR, R.; BOULANGER, P. Digital marbling: a multiscale fluid model. *Visualization and Computer Graphics, IEEE Transactions on*, v. 12, n. 4, p. 600–614, July 2006. ISSN 1077-2626. Citado 2 vezes nas páginas 8 e 10.

17 SIMPSON, T. W. et al. Kriging Models for Global Approximation in Simulation-Based Multidisciplinary Design Optimization. *AIAA Journal*, v. 39, n. 12, p. 2233–2241, 2001. Citado na página 8.

18 ANDO, R.; TSURUNO, R. Vector graphics depicting marbling flow. *Computers & Graphics*, v. 35, n. 1, p. 148 – 159, 2010. ISSN 0097-8493. Extended Papers from Non-Photorealistic Animation and Rendering (NPAR) 2010. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0097849310001718>. Citado 2 vezes nas páginas 8 e 10.

19 GROSSMAN, R. *Digital Painting Fundamentals with Corel Painter 11*. 1st. ed. USA: Course Technology PTR, 2009. ISBN 1598938939, 9781598638936. Citado na página 9.

20 BODDY-EVANS, M. *Corel Painter 9 Tutorial: Marbling*. 2006. <http://painting.about.com/od/digitalart/ss/Painter9_Marble.htm#step1>. [Online; accessed 2016-04-29]. Citado na página 11.

21 KUZMIN, D. *Introduction to Computational Fluid Dynamics*. 2010. <http://www.mathematik.uni-dortmund.de/~kuzmin/cfdintro/cfd.html>. [Online; accessed 2015-06-20]. Citado na página 13.

22 PEIXOTO, P. da S.; RODRIGUES, W. G. *Noções de Mecânica de Fluidos com Aplicações em Perfis Aerodinâmicos*. 2009. <http://www.ime.usp.br/~pedrosp/CFD.pdf>. [Online; accessed 2016-04-20]. Citado 2 vezes nas páginas 13 e 16.

23 ANDERSON, J. et al. *Computational Fluid Dynamics an Introduction*. 3rd. ed. [S.l.]: Springer-Verlag, 2009. Citado na página 14.

24 BRIDSON, R. *Fluid simulations for computer graphics*. 1st. ed. [S.l.]: A K Peters, Ltd., 2008. Citado 4 vezes nas páginas 14, 16, 29 e 30.

25 VARSHNEY, A. *Fluid Simulation Overview*. 2009. <http://www.cs.umd.edu/class/fall2009/cmsc828v/presentations/Fluid_Sim_Overview.pdf>. [Online; accessed 2016-04-20]. Citado 2 vezes nas páginas 16 e 20.

26 BAKKER, A. *The Colorful Fluid Mixing Gallery*. 2006. <http://www.bakker.org>. [Online; accessed 2015-04-26]. Citado na página 17.

27 COMSOL. *Fluid-Structure Interaction (FSI) using COMSOL Multiphysics* . 2013. <https://br.comsol.com/blogs/fluid-structure-interaction-fsi-using-comsol-multiphysics/>. [Online; accessed 2015-11-21]. Citado na página 17.

28 PHAGOR. *Simple Fluid Dynamics Sketch*. 2013. <https://www.openprocessing.org/sketch/197>. [Online; accessed 2016-06-20]. Citado 8 vezes nas páginas 25, 29, 46, 47, 48, 49, 50 e 53.

29 WOLFF, D. *OpenGL 4 Shading Language Cookbook*. 2nd. ed. [S.l.]: Packt Publishing, 2013. Citado na página 27.

30 SY2002. *Syfuid*. 2014. <http://www.sy2002.de/>. [Online; accessed 2016-06-30]. Citado 15 vezes nas páginas 29, 31, 32, 34, 36, 37, 38, 40, 41, 42, 44, 45, 48, 51 e 58.

31 INC., T. K. G. *OpenGL ES Extension*. 2013. <https://www.khronos.org/registry/gles/extensions/OES/OES_gpu_shader5.txt>. [Online; accessed 2016-12-20]. Citado na página 49.

32 OSHER, S.; RUDIN, L. I. Feature-oriented image enhancement using shock filters. *SIAM J. Numer. Anal.*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, v. 27, n. 4, p. 919–940, ago. 1990. ISSN 0036-1429. Disponível em: <http://dx.doi.org/10.1137/0727053>. Citado na página 52.

33 CORPORATION., N. *Chapter 27. Advanced High-Quality Filtering*. 2016. <http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter27.html>. [Online; accessed 2016-11-15]. Citado 2 vezes nas páginas 52 e 53.

34 FLICKR, a. Y. c. *Marbleize textures*. 2016. <https://www.flickr.com/photos/clicksnappy/galleries/72157625023717413/>. [Online; accessed 2016-12-22]. Citado 2 vezes nas páginas 56 e 57.

35 PEARCE, S. C. *Wet and wild: my first crack at marbling paper*. 2013. <http://paperwithapast.blogspot.com.br/2013/01/wet-and-wild-my-first-crack-at-marbling.html>. [Online; accessed 2017-01-10]. Citado na página 65.

36 CERRÉ. *Suminagashi*. 2013. <http://http://2or3things.blogspot.com.br/2011_11_01_archive.html>. [Online; accessed 2017-01-10]. Citado na página 65.

37 WEERD, P. D. Perceptual filling-in: more than the eye can see. In: MARTINEZ-CONDE S.L. MACKNIK, L. M. J.-M. A. S.; TSE, P. (Ed.). *Visual PerceptionFundamentals of Vision: Low and Mid-Level Processes in Perception*. Elsevier, 2006, (Progress in Brain Research, v. 154, Part A). p. 227 – 245. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0079612306540129>. Citado na página 70.

38 LEM, B. *Marble Spheres 3d Abstract*. 2015. <http://fineartamerica.com/featured/marble-spheres-3d-abstract-bluedarkart-lem.html>. [Online; accessed 2017-01-23]. Citado na página 74.

39 LIBRARIES, U. of W. *Marbled Paper Patterns*. 2017. <http://content.lib.washington.edu/dpweb/patterns.html>. [Online; accessed 2017-01-11]. Citado na página 75.