Dynamic Paint for Maya
Creation and Use in the Animation Short Color Clash

Department of Computer Science
California Polytechnic State University, San Luis Obispo
Christopher Wallis
June 12, 2013

Abstract
This write-up describes the creation and utilization of a paint effect from both the perspective of the programmer and the animator. The paint effect simulates a 2D fluid using the algorithm described in Jos Stam’s “Real-time Fluid Dynamics for Games” [4], bakes the results into an image sequence, and then wraps the sequence around a 3D mesh. The second half of this paper describes how the effect was integrated into the animation short, “Color Clash” and the animation pipeline that was used.

© 2013 Christopher Wallis
1 Introduction

Maya, a 3D graphics software used for modeling and animation, contains many various physics simulations, but lacks the ability to dynamically paint a mesh. Dynamic paint is the ability for some mesh to be able to “paint” or alter the texture of another mesh, typically via collisions. Being able to dynamically paint a mesh can be creatively used to create a range of effects such as paint splatters on bricks walls, footprints in snow, ripples in water, etc.

![Blender’s Dynamic Paint being used with a rain particle system to make the plane look wet][3]

![Blender’s Dynamic Paint being used with a character’s hand to give the appearance of dusting off a mirror][3]

2 Related Works

Blender, the free open-source competitor to Maya, already has dynamic paint as a physics module. At the time of this paper, the dynamic paint module in Blender offers several types of paint physics including “spread”, “drip”, and “shrink.” Although “spread” and “shrink” can be used for interesting effects, they are not truly fluid simulations and have limited parameters that can be used for customizing effects. Drip is a full fluid simulation but when paint is drawn on top of another color, the top color dominates the bottom color, rather than a mix of the two colors. This is undesired when trying to fully emulate the properties of paint.
The primary problem is that effects created in Blender cannot be easily translated to Maya. Maya and Blender have multiple differences, particularly with how they handle physics-specific entities such as particle systems and fluids. As such, an artist cannot simply switch between using Maya and Blender when trying to create a particular effect. This can be problematic because Blender still sees a significantly lower adoption rate compared to Maya. In 2004, “90 percent of film effects houses and half of the North American game studios [were] using Maya in their projects.” [1]

3 Dynamic Paint for Maya

3.1 Overview

The first half of this paper discusses how a dynamic paint plugin was created for Maya. The plugin uses Maya’s C++ API and Maya’s built-in scripting language, MEL.

The first step is creating a 2D fluid simulation that imitates the behavior of paint. The fluid simulation algorithm is taken from Jos Stam’s ”Real-time Fluid Dynamics for Games.” The algorithm is extended for support for colors
and a mixing behavior that uses heuristics based on the behaviors of paint.

The 2D fluid simulation is then “wrapped” around a mesh and the fluid simulation is treated as a texture. By checking for intersections between the mesh and a particle system and transferring those intersection points from world space to UV space, the fluid simulation can then spawn paint at the corresponding coordinate.

Finally, each frame of the fluid simulation can be baked out into a sequence of texture maps and bump maps so that the physics calculations can be cached.

3.2 Fluid Simulation

The fluid simulation borrows from Jos Stam’s ”Real-time Fluid Dynamics for Games.” The implementation described by Stam requires no external dependencies allowing for easy use within Maya’s C++ API. In addition, the real-time design of the simulation keeps the bake times of the fluid simulation low.

Without going into too much detail of Stam’s implementation, Stam treats a fluid as a 2D grid of densities and velocities. In addition, the movement of the fluid is handled as if each cell in the grid were a particle. This behavior is taken advantage of in the extension to a fluid system with support for multiple colors.

Figure 4: Visual of Stam’s fluid simulation in a 2D grid[4]
3.2.1 Color Mixing Algorithm

Color mixing can vary depending on the artistic medium involved. Light mixing is additive and in theory, we like to think paint is subtractive. However, when mixing paint, the behavior is rarely that of a perfectly subtractive system. For example, when mixing to create a vibrant purple, one might expect to be able to just mix equal parts red and blue. Often the actual result will often end up being a dark, duller purple than what one might desire. This particular behavior, the tendency for paint to become more dull as more colors are mixed together, is used as a heuristic for colors mixing.

The actual color mixing algorithm is straightforward. When two colors are combined, linear interpolation is used with their corresponding densities to determine a final color:

\[
endColor = \frac{d_1}{d_1 + d_2}c_1 + \frac{d_2}{d_1 + d_2}c_2
\]

The result of the above algorithm when run overtime on a simulation with multiple colors mixing together will result in a gradual approach towards gray.

Figure 5: Various colors were created on the corners of the fluid simulation and then mixed in the center to form a brown color.
3.2.2 Moving to 3D

To move to 3D, the 2D fluid simulation is treated as a texture of a mesh. Maya’s MEL scripting environment is used to create a callback that is called every time a particle system collides with the mesh. The intersection point can then be converted from world coordinates to the mesh’s UV coordinates. These UV coordinates can then be used to tell the fluid simulation where it should paint.

This method does make assumptions on how a mesh is UV unwrapped. Each point on the mesh is assumed to map to a unique UV coordinate (i.e. no two distinct points on a mesh ever map to the same UV coordinate). This is somewhat undesirable since being able to map different coordinates to the same UV can allow for efficiency tricks such as having symmetrical sides of a mesh map to the same section on a texture.

3.2.3 Bump Map

Bump maps are a technique used to make a mesh appear to have fine details without adding more geometry. This is used by saving heights into a texture and using these values to calculate shading per pixel.

Because the simulation is only a 2D fluid simulation, wrapping the simulation around the mesh by itself is unconvincing and has a flat appearance. To create the effect of paint adding an extra layer onto the mesh, bump maps are generated based off the density values of the fluid simulation.
3.2.4 Baking

Physics calculations tend to be expensive and make it difficult to view the effect in real-time for quick feedback. The general solution is “baking” the effect - caching the calculations to a file so that they only need to be calculated once, and then can be efficiently read back from a file.

Keeping with this mentality, Dynamic Paint bakes out a texture map and bump map of the fluid simulation at each frame. The images can then be loaded into Maya as an image sequence that requires no recalculation of fluid physics.

3.2.5 Image Painting

Interesting effects such as burning or freezing a mesh can created by painting with an image rather than colors. This uses a similar method as the bump
map generation. Colors again are ignored and densities are used to determine how transparent an image is at each texel.

![Image](image.png)

Figure 8: Image painting was used in “Color Clash” to maintain the cloth-like textures and introduce noise to the paint for an added level of realism.

### 3.2.6 Extensions/Future Work

Currently, this implementation only supports the ability to choose a particle system as a “brush”. Blender offers support for many other types of “brushes” such as animated meshes and smoke simulations. In addition, Blender also allows several other interaction parameters such as using proximity to trigger paint rather than physical collisions.

The fluid simulation is hindered by many quirks of a UV-coordinate system. A UV unwrapping can allow for one UV coordinate to map to multiple parts of a mesh, which can lead to awkward results when using Dynamic paint. In addition, there is no connectivity between UV patches. Dynamic Paint would be much more suited for a system that does keep track of connectivity such as PTex.
4 Animation Pipeline

4.1 Motivation

Dynamic Paint for Maya was used to create an animation short titled “Color Clash.” The animation short takes place in a black and white world where two wizards end up battling each other and subsequently introducing color to the world. The color is added using dynamic paint to create the effect that these wizards are “painting” the world. This section gives an overview of the workflows followed in creation of the animation short.

![Figure 9: Example scene from “Color Clash”](image)

4.2 Pipeline Stages

The animation pipeline is split into 3 stages: pre-production, production, and post-production. [2] Pre-production introduces the creation of the story, storyboarding, and initial visualization of scenes. Production is the creation of 3D assets and animation. Post-production is the synchronization of audio and combining and modifying clips to create a cohesive flow.
4.3 Pre-Production

4.3.1 Storyboard

A storyboard was constructed to be able to visually explain the story, and the nature of the colors in "Color Clash." In addition it created a requirement to consider important factors in animation early such as camera angles and character poses.

4.3.2 Storyreel

Sketches were drawn out on flashcards to allow for an easy re-arrangement of scenes and quick creation of new scenes. The flash cards were scanned and put together with rough timing to create approximate timings that could help demonstrate the pace of the animation short.

4.4 Production - Modelling/Texturing/Binding

4.4.1 Modelling-Texturing-Binding Pipeline

Ideally, modeling, texturing, and binding would like to be thought of as sequential steps in creating an rigged model, but the reality is that this is a cyclic iterative process. Frequently errors in a mesh’s geometry will not be discovered until the binding stage, which requires one to go back to the modeling stage. These changes in the model will frequently mean the texture then also needs to be changed as well. As a result, it was found to be better to work on each phase in small iterative steps rather than long committed steps.
4.4.2 Modelling

Modeling was done entirely in Maya using box modeling with polygons. Character profiles were drawn on paper and then scanned into Maya and used as references when modeling. Because cloth simulation was a big part of “Color Clash”, the character and cloth were modeled separately.

4.4.3 Texturing

All textures were created in Photoshop. Resources were primarily real life textures (i.e. zoomed-in images of fabric) with color overlays applied.

A big problem with character textures was that they looked too matte and plain even with bump maps. This was solved by using noise and imitating signs of wear. Blurred pictures of autumn leaves were overlayed over textures to create varied coloration and a dirt Photoshop brush was applied afterward to show wear.
4.5 Production - Character Animation

4.5.1 Rigging

A rig was created for each character to make animation simpler. In addition, a small expression panel was created for simple alterations of the eyes such as anger, blinking, or squinting. The rig additionally had “key driven” attributes that would coordinate multiple character body parts with one value. This saved time when dealing with synchronizing the flapping wings of the butterfly, closing fingers, and symmetric eye expressions.

Figure 12: Example showing the rig of the yellow wizard. The expression panel is shown to the wizard’s right.

4.5.2 Body Movement

A recording of people acting out each scene was taken so that the small human movements could be observed and incorporated into the short. In addition, certain movements were exaggerated based on Disney’s principles of animation. Particularly, arcs and drag are taken into account in all of the animations.

4.6 Production - Physics Animation

4.6.1 Dynamic Paint

The Dynamic Paint was incorporated into the animation short by using the 3D fluid simulation to hide a simple particle system that does the actual work of triggering the paint.
4.6.2 Maya’s Physics Simulations

Maya’s cloth simulation is used for the robes of both of the wizards as well as the sign poster that displays the short’s title.

The fur simulator is used to create the grass effects. The trees use Maya’s paint effect, which offers the option to specify a turbulence as a fast way to imitate wind. The fireballs and smoke both use Maya’s fluid simulation.

The rainbow particle system following the butterfly in the ending scene is a particle system that uses a MEL script that takes the particle id and maps it to a color range. Because particle id correlates to when the particle is emitted, the mapping is a straightforward iteration over the color wheel.

4.7 Post-Production

4.7.1 Adobe After Effects

Maya offers the ability to do special camera effects such as motion blur and depth of field. The handling for each of these effects are done with physical accuracy that can be costly and is often unnecessary. Adobe After Effects
is able to do most of these effects with comparable aesthetic appearance and with a significantly lower computational cost. Further, After Effects allows the ability to play with different settings without needing to re-render the entire scene. For tweaking small details such as color correction and contrast, After Effects is significantly more efficient than changing settings within Maya.

4.7.2 Windows Movie Maker

After processing each scene within After Effects, the scenes are then concatenated together in Windows Movie Maker. For transitions that seemed to be jumpy, a blur-in transition was applied to maintain the cohesion in the story. This is can be noticed in the walking scene and after the hand-shaking scene.

5 Conclusions

Creating an animation short allows the opportunity to experiment with many of the roles within the animation pipeline. Even within software developer roles, the creation of an effect requires one to work on multiple levels of abstraction. The mindset of working with Maya’s C++ API differs from that of working with the MEL scripting environment. Particularly, working in C++ encourages the developer to aim for long-term solutions, whereas MEL is better suited for quick-fixes.

As a whole, the project was run on a low budget. The free student edition of Maya was used for the modelling and animation. Free 30-day trials were used for Photoshop, CrazyBump, and After Effects. Additionally Windows Movie Maker is free with Windows 8. Most learning was done with free online resources with the exception of Lynda.com, which requires a $25 monthly fee. The implication is that CG animation is no longer a field that is limited to industry or film schools. It is easily imaginable that a small team could produce a high quality CG animation short within the span of several months with little to no funding.
5.1 Acknowledgments

Thanks to Zoe Wood, my senior project advisor and the computer graphics professor at Cal Poly for believing I would be able to complete an ambitious project.

Thanks to Enrica Costello, an art professor at Cal Poly for giving feedback on my animation short ideas and general tips for working with Maya.

Finally, thanks goes out to Ryan Waczek, a music student at Cal Poly, for producing the wonderful sound and music.
References


Appendices

A Maya Command code

Maya’s C++ API allows you to create a command that can be called within Maya via its MEL terminal. Below is a slimmed down version of the code required to make a basic command.

class DynamicPaintCmd : public MPxCommand
{
    public:
        DynamicPaintCmd();
        virtual ~DynamicPaintCmd();
        static void* creator();
        bool isUndoable() const { return false; }
        MStatus doIt(const MArgList&);
        MStatus undoIt() { return MStatus::kFailure; }
    
};

DynamicPaintCmd::DynamicPaintCmd() {}

DynamicPaintCmd::~DynamicPaintCmd() {}

void *DynamicPaintCmd::creator()
{
    return new DynamicPaintCmd();
}

MStatus DynamicPaintCmd::doIt( const MArgList& args ) {
    // Argument parsing and logic goes here
    return MS::kSuccess;
}

MStatus initializePlugin(MObject obj)
{
MStatus status;
MFnPlugin plugin(obj, "Chris Wallis", "1.1");
status = plugin.registerCommand("dynamicPaint",
    DynamicPaintCmd::creator);
return status;
}

MStatus uninitializePlugin(MObject obj) {
    MStatus status;
    MFnPlugin plugin(obj);
    plugin.deregisterCommand("dynamicPaint");
    return status;
}

B Fluid Simulation Header

The fluid simulation was abstracted away from any Maya interaction or dependencies. Below is the header for the FluidSimulation class that drives all the fluid calculations for Dynamic Paint.

class FluidSimulation {
    public:
    FluidSimulation(int width, int height);
    ~FluidSimulation();
    SmokeParticle *getDensity();

    void addVel(int x, int y, int brushRad, bool outward);
    void addDens(int x, int y, int brushRad, uchar r, uchar g, uchar b);
    void addBound(int x, int y, int blockSize);

    void setUniformForce(float u, float v);

    void update(float dt);
    void dens_step(float dt);
    void vel_step(float dt);
}
void setSurfaceFriction(real friction) { frictionVal = friction; }

int getSize() { return size; }
VectorField getUniformField() { return VectorField(uniform_u, uniform_v); }
const bool *getBounds() const { return bounds; }

void addDensity(int x, int y, float amt, uchar r, uchar g, uchar b);
void addVelocity(int x, int y, float xAmt, float yAmt);

void reset();

//Private variables
}

C Color Mixing Code

The code below is the linear interpolation between two particles based upon density values.

inline static color_t lerpColor(const SmokeParticle &sp1,
const SmokeParticle &sp2) {

float totalDens = sp1.density + sp2.density;
float weight1 = sp1.density / totalDens, weight2 = sp2.density / totalDens;
  // Add .5f so the value is rounded up when casted to a uchar
  ret.r = (uchar) (weight1 * sp1.getR() + weight2 * sp2.getR() + .5f);
  ret.g = (uchar) (weight1 * sp1.getG() + weight2 * sp2.getG() + .5f);
  ret.b = (uchar) (weight1 * sp1.getB() + weight2 * sp2.getB() + .5f);
}

return ret;
}

// The operator+ is overwritten to automatically apply the color linear interpolation, so that the SmokeParticle can directly substitute a float in Stam’s fluid simulation code
SmokeParticle operator+(const SmokeParticle &o) const {
    color_t t = lerpColor(*this, o);
    return SmokeParticle(density + o.density, t.r, t.g, t.b);
}

D MEL script (Without GUI code)

The Maya C++ command is then called via the MEL script below. The
script below also handles retrieving the collision points from a particle system
hitting a mesh, and converting them to a UV coordinate on the mesh. The
GUI code is not listed for brevity.

loadPlugin "nearestPointOnMesh.mll";

string $g_psList[];
vectors $g_colorList[];

proc setPSCollisionCallback(string $ps, string $mesh, string $onCollision) {
    connectDynamic -collisions $mesh $ps;
    event -name mainEvent -die 0 -count 0 -proc $onCollision $ps;
}

proc removePSCollisionCallback(string $ps) {
    event -d -n mainEvent $ps;
}

// Returns an array with the index 0 holding the u-coordinate,
// and index 1 holding the v-coordinate
proc float[] getNearestUV(string $mesh, vector $p) {
    string $npom =
        'nearestPointOnMesh -ip ($p.x) ($p.y) ($p.z) -u -v -p $mesh';
    float $uvCoord[];
    $uvCoord[0] = 'getAttr ($npom + ".u")';
    $uvCoord[1] = 'getAttr ($npom + ".v")';
    return $uvCoord;
}
global proc UVPrintEvent (string \$ps, int \$psId, string \$mesh) {
    vector \$partPos = 'particle -attribute position -id \$psId -q \$ps';
    float \$uvCoord[] = getNearestUV(\$mesh, \$partPos);
};

global proc UVPaintEvent (string \$ps, int \$psId, string \$mesh) {
    global int \$g_frames;
    global string \$g_canvasName;
    global string \$g_psList[];
    global vector \$g_colorList[];
    vector \$clr;
    vector \$partPos = 'particle -attribute position -id \$psId -q $ps';
    float \$uvCoord[] = getNearestUV(\$mesh, \$partPos);
    string \$psName = 'substitute "Shape" \$ps ""';
    int \$i; 

    // Find the particle systems brush color
    for (\$i = 0; \$i < size(\$g_psList); \$i++) {
        if ('strcmp \$g_psList[\$i] \$psName' == 0) {
            \$clr = \$g_colorList[\$i];
            break;
        }
    }

    // If the particle system isn't a brush, then paint on the canvas
    if (\$i == size(\$g_psList)) { error (\$psName + " not found"); return; }

    \$clr = <<\$clr.x * 255, \$clr.y * 255, \$clr.z * 255>>;
    dynamicPaint -b \$uvCoord[0] \$uvCoord[1] \$clr \$g_canvasName;
};

global proc bakePaint(int \$startFrame, int \$endFrame, string \$ps[],
    vector \$colors[], string \$mesh, string \$canvasName,
    string \$dir) {
    global string \$g_psList[];
}
global vector $g_colorList[];
global string $g_canvasName;

int $i;

$g_psList = $ps;
$g_colorList = $colors;
$g_canvasName = $canvasName;

if ($startFrame < 0 || $endFrame < 0) {
    error "start and end frames must be positive values";
    return;
} else if ($startFrame >= $endFrame) {
    error "Start frame must be less than the end frame";
    return;
}
waitCursor -state on;

currentTime $startFrame;

for ($i = 0; $i < size($ps); $i++) {
    setPSCollisionCallback($ps[$i], $mesh, "UVPaintEvent");
}

int $frameOffset;
for ($frameOffset = 0; $startFrame + $frameOffset <= $endFrame;
    $frameOffset++) {
    currentTime ($startFrame + $frameOffset);
    string $fileName = $dir + "\frame." + $frameOffset + ".bmp";
    string $bumpName = $dir + "\bump." + $frameOffset + ".bmp";
    dynamicPaint -u .04 $canvasName;
    dynamicPaint -w $fileName $bumpName $canvasName;
}

for ($i = 0; $i < size($ps); $i++) {
    removePSCollisionCallback($ps[$i]);
}
currentTime $\$startFrame;
waitCursor -state off;
}