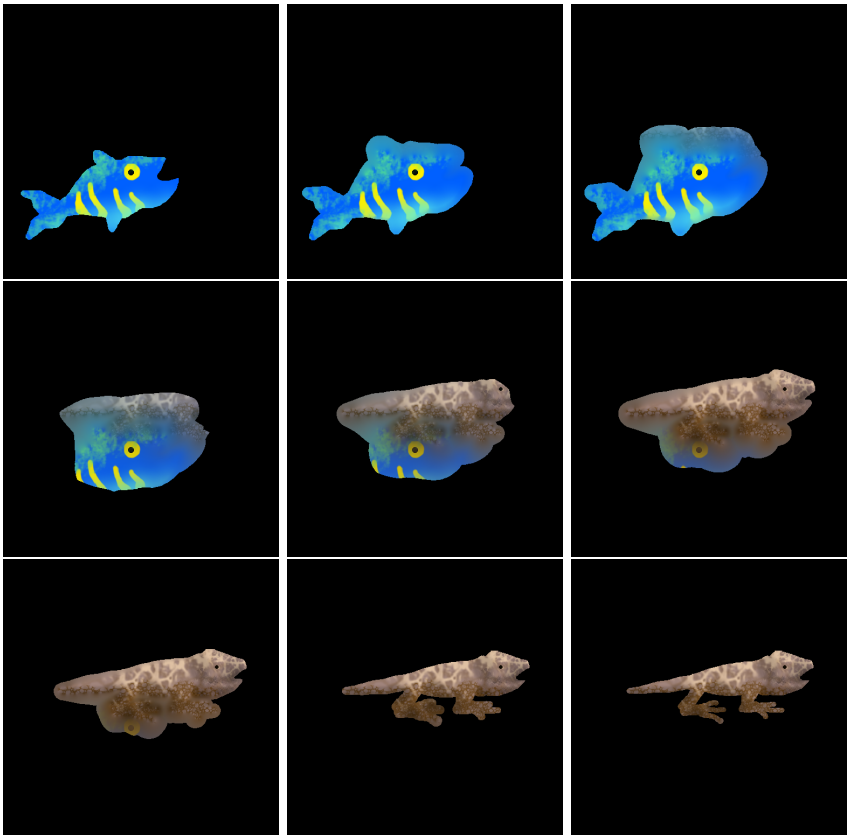
2D Image Space-Time Blending Using Textures

Felix Marrington-Reeve

23rd January 2017



Supervisor: Valery Adzhiev

Contents

1 Abstract

This report presents a novel technique for space-time blending of heterogeneous objects with textures, based on the work of Pasko et al. It deals with creation of the objects and blending of textures using methods that are either new or newly combined. The report assesses a common problem of creating in between images automatically in order to blend between 2 target images, and analyses the success of the methods used. Applications of this technique include 3D coloured model blending and automatic interpolation for 2d images.

2 Introduction

Heterogeneous object metamorphosis is a complex problem in animation. While there are a number of methods to creating a blend between two 2D shapes, many methods have disadvantages such as polygonal correspondence or requiring shapes to be defined by strokes rather than something more intuitive. Techniques that allow for blending without these restrictions tend not to allow for complex attributes such as textures. The aim of this paper is to research and develop a method for blending between two images to create a series of inbetween images which allows for attribute blending; in this case two textures, one applied to each keyframe, which can be blended between.

In [9], Pasko et al. put forward a method for blending between shapes that can be applied to both 3D and 2D spaces, and requires no correspondence in topology or position for the metamorphosis to work. In cases where such assumptions are not required, attributes such as colour are still difficult to blend between, particularly in areas of the output where the initial shapes do not overlap. A similar problem is found in automatic 2D inbetweening, where it is often not enough for images to be defined by 2D pixels which are often used to represent a 3D space [2].

Even once the metamorphosis of shape is calculated, there is very little research on bending between other attributes of the objects, such as colour. If the 2 input images or keyframes have textures on them, there are currently no methods for blending between arbitrary raster images. This report presents a method for blending between 2 images with textures applied to them.

3 Background and Alternative Approaches to Metamorphosis

3.1 Image Prediction

Motion estimation can be relatively easily achieved using polynomial expansion [3]. The OpenCV library provides some open source functions that allow motion flow to be calculated and used for naive image prediction, such as the next frame or in between frames *Fig 1.*

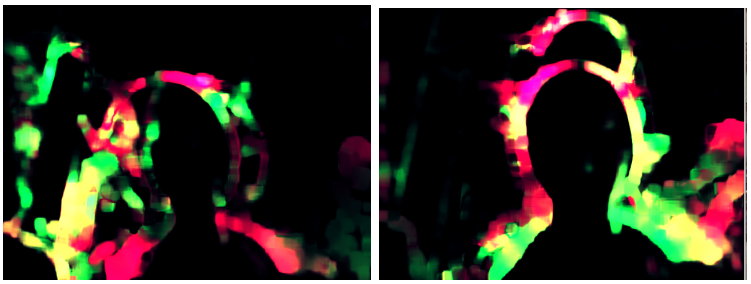


Figure 1: Dense Optical Flow tested on a webcam, visualised with Red and Green values

```
Step 1:
        Use OpenCV to calculate a dense optical flow field.
Step 2:
        Use the flow result to predict the next frame:
        Copy the 2nd frame used in prediction
        For each pixel:
                Read flow vector of the pixel
                Copy the pixel to a position offset by the
                  flow x and y values
                Return the newly created frame
```

This approach works fairly well on simple images with basic motion and can be used to get inbetween frames or predict the next frame.

Frame prediction and inbetweening was very effective in the case of *Fig 2*, in which the colours, shapes and movement are very simple. Although the algorithm is basic, in specialised cases this may be suitable for interpolating or predicting frames. As far as I know this is the only program of this form, and as such provides an innovative alternative to rendering new frames if the animation is very limited.

However, due to its simplistic nature, it creates artefacts on more complex images and motions, since as parts move out of the way, they reveal areas where new information needs to be filled in and as such is not appropriate for the aim of this project, such as in *Fig 3*.

3.2 Texture Synthesis / Wave Function Collapse

One Possible solution to the issue of missing information is to use the images to create possible or probable replacements for areas of the image with no information. [4] synthesises textures using a number of techniques in which adjacency rules and transfer techniques are used to create plausible Restoration of a number of images. This allows images with sections missing to be filled in with plausible colours.

Some of these ideas are implemented in the program, *WaveFunctionCollapse* [8], which uses ideas from quantum physics to create an algorithm that

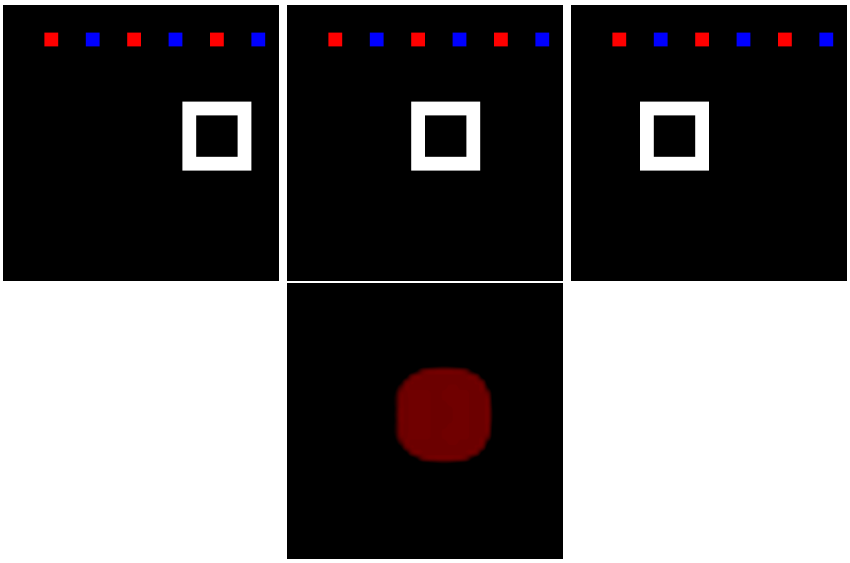


Figure 2: (Above) Horizontal movement of a simple square sprite, frames 1 and 2, and a predicted frame, 3. (Below) Dense motion flow analysis of the movement, visualised so that Red indicates horizontal movement

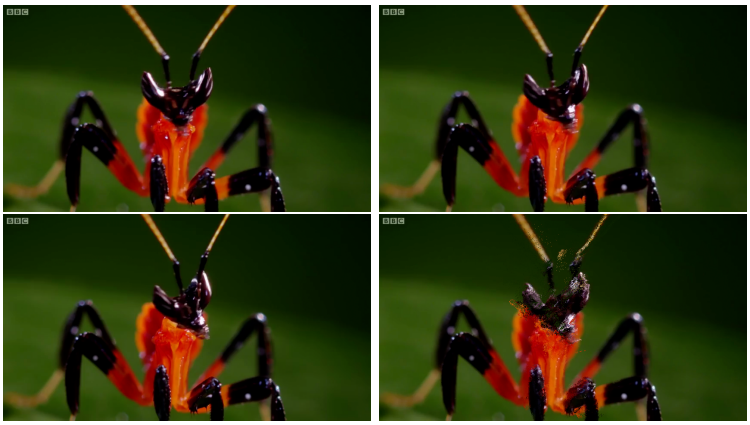


Figure 3: (Above) frame 1 and 2 of an insect moving its head. (Below Left) frame 3 of the insect's movement. (Below Right) frame prediction of the insect's movement

can generate *locally similar* images to input images. By using a similar algorithm with a 3D array of pixels from a video, where the third dimension is time, it might be possible to get *locally similar* output frames as predictions or automatic inbetween images. Using this large 3d array, the program might be able to make more advanced predictions based on more than just the previous 2 frames.

Step 1: Read an image sequence and create a 3D array of pixel

```
        colour and flow
Step 2: Read array values and create data structure suitable
        for wave function collapse
Step 3: Synthesise the next frame in the sequence based on
        likely flow and colours in the desired frame
```

This approach, while theoretically interesting, will likely not produce any accurate results, as the wave function collapse algorithm due to its success at *locally similar results*, rather than results that make logical sense in a wider context such as the whole frame or sequence. However, it is a new method that has not been explored before and might make for interesting future research.

3.3 Alternative Approaches

An early example of inbetweening 2D images can be found in [2], where the authors present a system for morphing lines into different shapes and creating inbetweens. However, the line strokes must have correspondence and a skeleton must be set up in order to fully utilise the system.

A more modern solution can be found in [5] present an approach for automatic in betweens that relies on creating a 3D model from a front-view and side-view of a character, and on having accurate drawings of the 2D character in order to create correspondence between strokes. The strokes can be incorrectly annotated and there is no option for transitioning between radically different images such as drawings with radically different strokes or shapes.

3.4 FRep Solids and Heterogeneous Objects

An implicit shape can be represented by either $F(x, y)$ for 2D shapes or $F(x, y, z)$ for 3D shapes, where on the defining curve or surface $F(x, y) = 0$ in 2D or $F(x, y, z) = 0$ in 3D. An FRep solid is defined by primitives and operations, where $F \leq 0$. In [Space-time blending] Pasko et al. put forward a method which converts two k dimensional input shapes into $(k + 1)$ dimensional shapes in order to blend between them in the higher dimension. This allows inbetween models that do not rely on matching topology of the two inputs, due to their functional representation [1]. In 2D:

```
Two shapes are given using the x,y plane
Each shape is used as a cross section of a half
  cylinder in 3D space, by extending them in the
  z dimension, with space inbetween for the blend
The blending union operation is applied, this can
  be user controlled with different parameters
Using the z dimension as time, take cross sections
  of the shape along the z axis to display a
  smooth blend between the 1st and second shape
```

The implementation in 3D is essentially the same, but raises the shapes into 4D and uses the 4th axis as time.

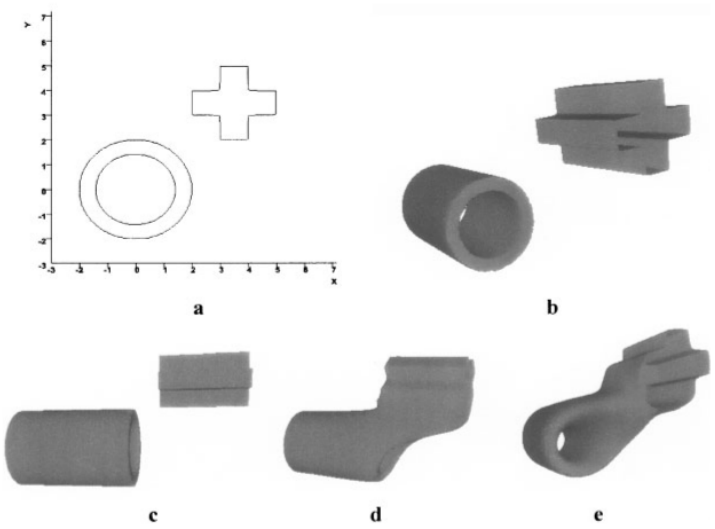


Figure 4: From [9], a visual example of how a 2D blend is achieved by raising the space to 3 dimensions

By blending using functional representation and this metamorphosis method, it is possible to create in between frames that have the following advantages over the other methods mentioned.

- The keyframes need not overlap or have similar shapes or colours
- Textures should be implemented and a method or method of blending between textures should be present, including areas where no texture information is present, such as outside of both keyframes shapes
- A relatively high level of user control should be possible

Heterogeneous objects which include both the shape and attributes of the object, such as colour, can also be represented using signed-distance fields. [12] states that implicit functions have uses in entertainment, engineering and the medical industry. The potential that multiphase implicit functions have is also communicated, with implicit modeling that can represent different internal materials of objects. This allows the internal structure of objects to be modeled using implicit functions, which has further uses. This adds complexity to the space-time blending operations which is addressed in [11] in which the authors propose interpolation of section attributes such as colour, as well as using blends with established correspondence between partitions and blends with no established correspondence. The authors also suggest a method for a summation of a voxel field, or a pixel array in 2D, for a weighted average of colours at the inbetween stage of the blend where colour is not explicitly defined.

The work of Pasko et al. is implemented in the following section with visual examples, followed by an implementation of the paper's theoretical method for blending between textures. Also included is a new alternative method for blending the colour, and a new combination of signed-distance field conversion with blending.

4 Development

I began to implement a similar set of 2D functions to the FRep program *HyperFun* [`http://www.hyperfun.org`] in my own program, using C++. This began with a simple program in which functions can be written in GLSL to create a boolean image of the signed distance field that it creates. This allows an input of time to blend between two functions using the methods put forward in [10], with a series of black and white output images.

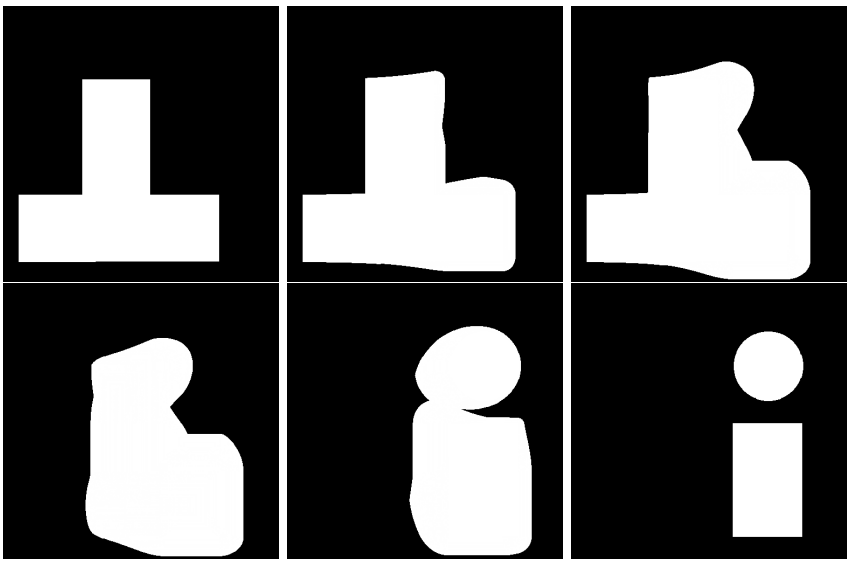


Figure 5: Blending between 2 boolean shapes

4.1 Single Colours

Colours can be blended between using a simple interpolation between the two initial states, where pixels colours are weighted by the absolute value of f_1 and f_2 [11]:

$$c(t) = w_1(t) \cdot c_1 + w_2(t) \cdot c_2$$

Where

$w_1(0) = 1$ and $w_2(0) = 0$ for $f_1 \leq 0$ (on or inside G1)

$w_1(1) = 0$ and $w_2(1) = 1$ for $f_2 \leq 0$ (on or inside G2)

$w_1 + w_2 = 1$

The interpolation uses initial constant colours, and as shown in *Fig 6*, the colour change is non-linear and dependent on the shape of the two input shapes. The colour of the object can also be represented as a function of space, or $c_1(x, y)$ and $c_2(x, y)$ to two procedural textures that can be blended between.

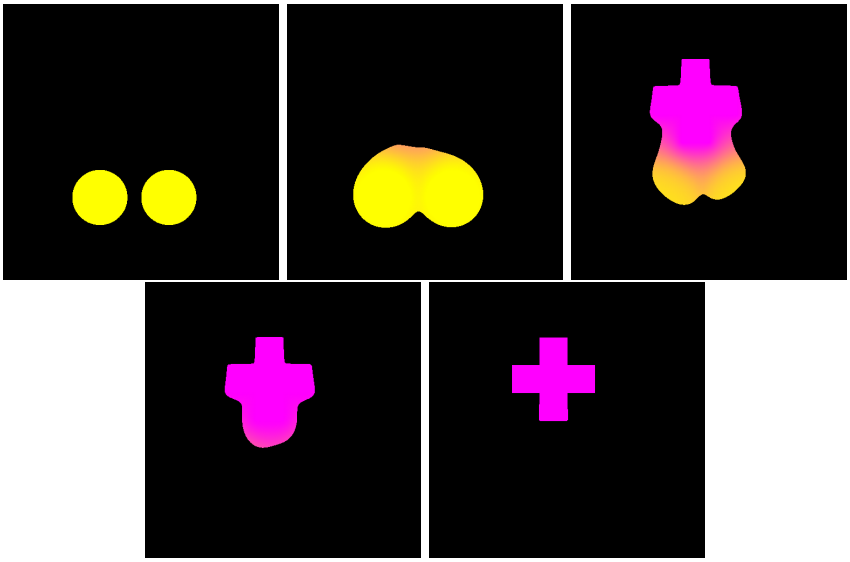


Figure 6: Blending between two yellow circles and a pink cross, in RGB colours space

The interpolation method should be dependent on the attribute being interpolated. In *Fig 7* an example is shown of interpolating using the HSV colour space, rather than RGB. The colours are less intuitive as they are calculated by blending in the HSV space, causing unusual hues to appear as the colour is interpolated. In most cases the RGB colour space is more appropriate, although the option to use HSV to interpolate the colours can produce visually interesting results if desired

4.2 Partitions and Texturing

Basic partitioning is possible using functional definitions for the colour of each object, using intersections of the original shape with sectional functions. There is no correspondence between partitions, and so the colour is calculated as follows, for the overall contribution of the object to the attribute at the given

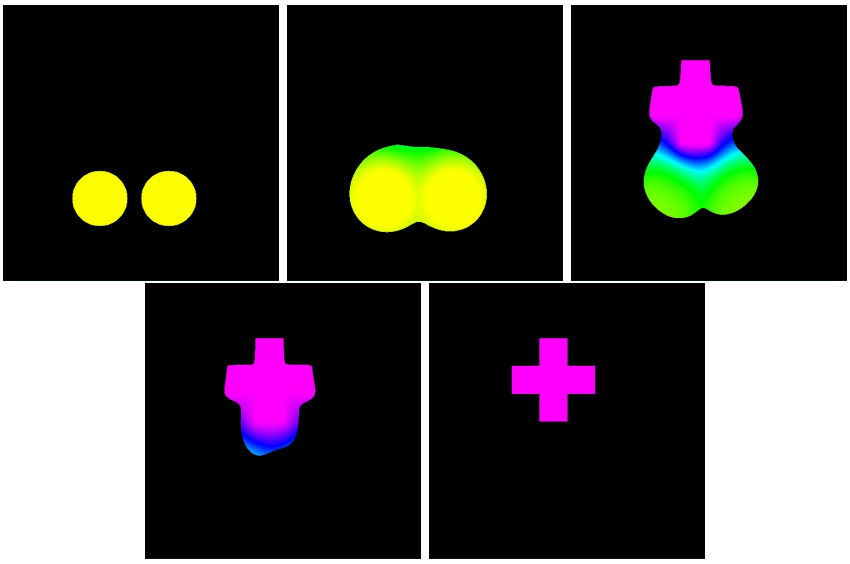


Figure 7: Blending between two yellow circles and a pink cross, in HSV colour space

point:

$$\frac{c_i(x) = \sum_{j=1}^{N} \tilde{w_j}(x)\tilde{c_j}}{\sum_{j=1}^{N} \tilde{w_j}(x)}$$

where N is a number of partitions and the weight of each partition is give by $\tilde{w_j}(x) = \frac{1}{f_j(x)}$.

Once this algorithm is implemented, it is relatively straightforward to implement a system where pixels or voxels of a provided texture can be used as numerous individual partitions, each with a given colour. The colours of the input textures ought to be defined only inside the input shapes. The colour of an output pixel is calculated using the distance to other pixels on the image, which allows a smooth blend between the two input textures. Although a similar method is suggested in [11] I believe this to be the first implementation of this method, and it works well in practice, as shown in *Fig 9*, although due to the complexity of the calculations and the fact that it is not currently running on the graphics card, it does not run in real time.

4.3 Alternative Colour Calculation Method

An alternative implementation of generating colour is to use the colour of the closest point where $f \leq 0$ for each object and blend the result. This looks odd looking at the inside of the object but might seem more intuitive for certain applications such as looking at the surface of a 3D object for example, and retains the detail of the original surface while the shape expands. This is a relatively simple approach but is of my own design and I believe could be seen

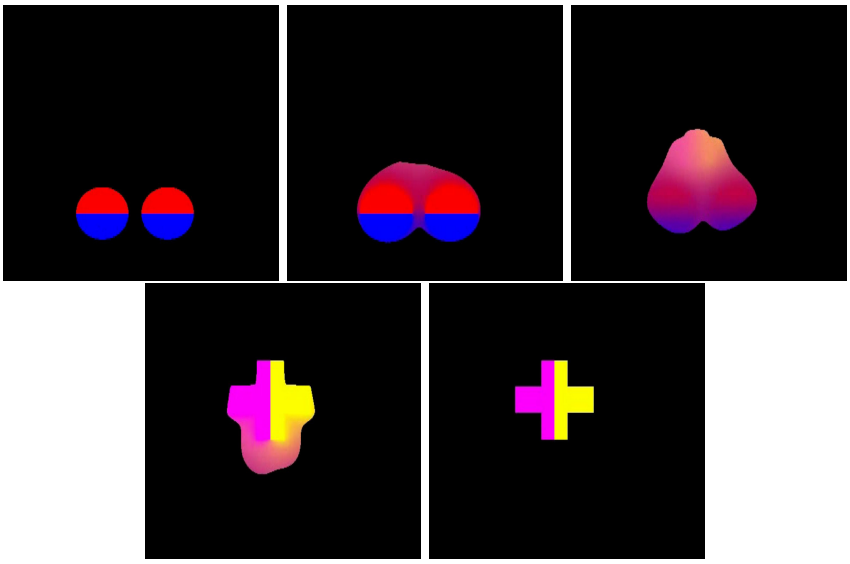


Figure 8: A blend involving two partitions in each input, with no correspondence between partitions

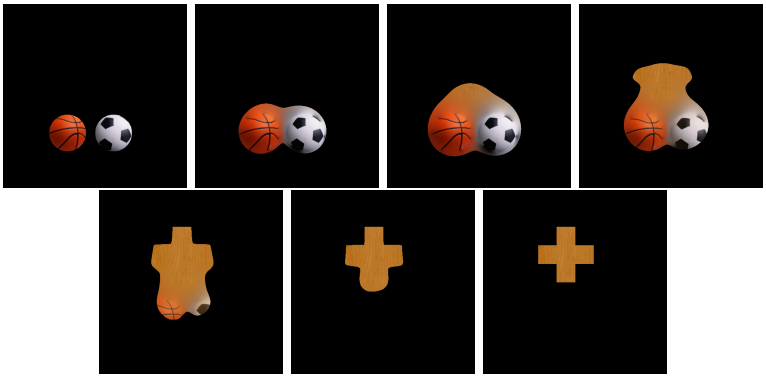


Figure 9: A blend between textured shapes, from two balls to a wodden cross.

as useful in some situations where the internal structure of the blend is less important than the surface colour.

Another option in the program is to use a lower resolution version than the texture in order to preview the blend. This allows for faster results from the colour calculation, although it is significantly less accurate with lower resolutions *Fig 11*, it can be used to get relatively high quality results with much faster calculation times.

Figure 10: A blend between textured shapes, from two balls to a wodden cross, using the closest value to get colour when outside of the initially defined states. Note that in frames 2 and 3 the colour at the surface is arguably fairly intuitive, especially if a similar technique were implemented in 3D.

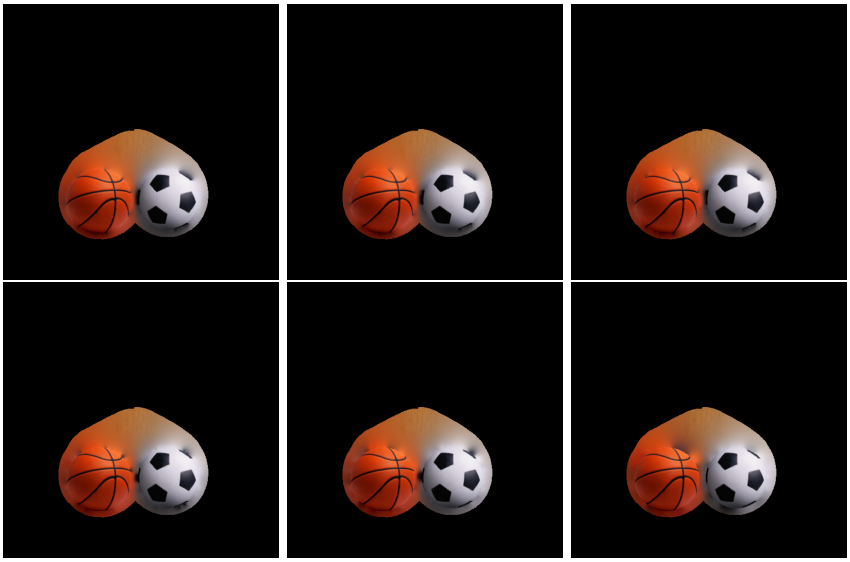


Figure 11: The blend using different resolutions of the texture to create blended colours, at whole resolution, then $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{5}$, $\frac{1}{10}$, $\frac{1}{15}$ resolution respectively. At $\frac{1}{2}$ and $\frac{1}{3}$ resolution, the images are likely to be detailed enough to be used, and offer a significant speed boost over the full resolution, in this case up to 10 times faster to calculate

5 Image to SDF Conversion

One area that Pasko et al. do not seem to have explored is the creation of Signed-Distance Fields from methods other than functions. As such, it is difficult to argue for the use of space-time blending, since it requires that shapes be defined by functional representations, which are often unintuitive without

specialist software to create them. The 8SSEDT or *8-Points Signed Sequential Euclidean Distance Transformation* algorithm from [6] is implemented, which takes a boolean raster grid and uses it to create a SDF.

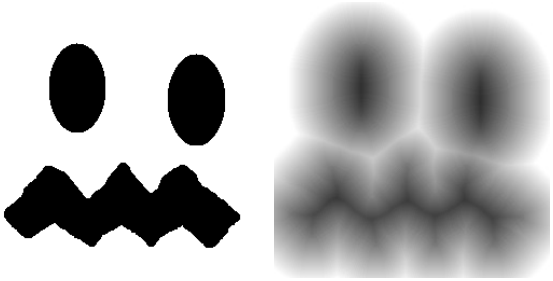


Figure 12: From Mitton [7], taking a boolean image and creating a signed-distance field

This allows the user to draw an image with a black background for *outside* the object and use textures of their choosing for *inside* the object. This method is implemented to allow a more artist-friendly appraoch to creating the two keyframes of the blend. This algorithm is easy to incorporate into an FRep model as instead of using functions to create the implicit shape, a $SDF(x, y)$ function can be implemented which can be used to look up values once the SDF is generated. The result of this is the ability to generate non-trivial blends more easily, and it the implementation is entirely new for space-time blending.

5.1 8SSEDT Implementation

The 8SSEDT algorithm is created by Leymarie, F. and Levine, M. D. [6] and improved by Richard Mitton [7]. The algorithm requires two grids which are then combined to make the signed distance field. The aim is to create 2 positive distance grids, which contain the distance to the surface of the surface. One contains distances on the inside, while the other contains distances on the outside. The two grids are initialised to either inside or outside and then propagated through to write in their x-distance and y-distance to the surface, by traversing the grid multiple times in order to compare their x-dist and y-dist to their neighbours and update them if the result is an *improvement*.

Points are defined as having a dx and a dy value, where dx/dy will contain the offset from this point to the nearest on the opposing side. They are initialised as *inside* or *empty* where *empty* = INF, INF, and *inside* = 0, 0. The grids are then traversed in order to propogate through and write their new values. The grids are then combined to create the signed field. This must be done to both images in order to blend between raster images. For more detail on the algorithm see [6] or the implementation in the appendix.

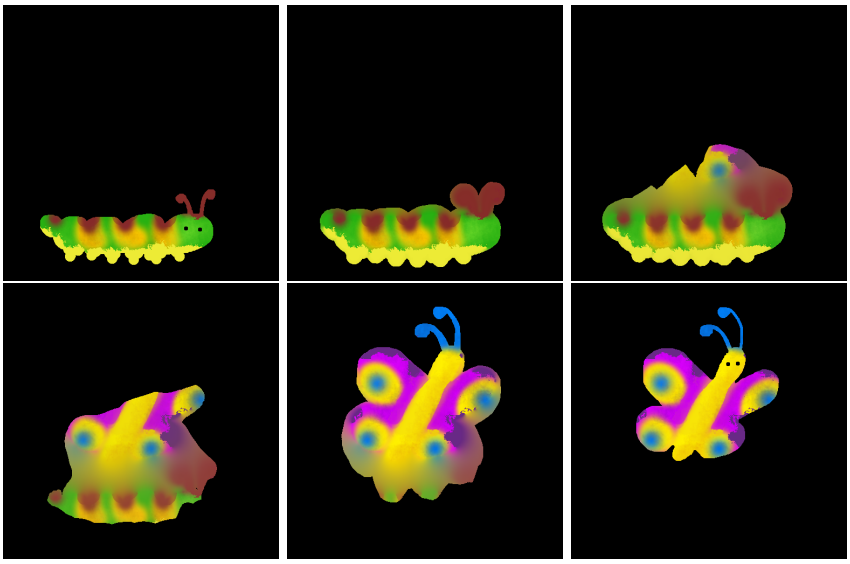


Figure 13: A caterpillar blending to a butterfly, created from 2 png input images

5.2 Further development

In order to further improve the system it would be beneficial to integrate the functional representation that is shown in the tests in the paper. It would also be beneficial to focus on other areas that were not touched upon in the project, such as optimisation of the colour averaging algorithm, such as using the grpahics card to speed up processing, or using a quadtree or similar data structure. It is also possible to implement the functionality in 3D to compare the results found there.

6 Conclusion

A method has been presented and implemented for allowing a blend between two raster images, that is not dependent on correspondence of attributes between them. It is the first implementation of the effect, and the first combination of converting images to signed-distance fields and then blending between them. The ability to use the closest point on the surface of the original shape is a new approach that, while simple, may have uses in 3 dimensions in particular in order to create a more intuitive surface of a model while it is going through space-time blending. The program is implemented in a user-friendly way that does not require functional representation of the shapes involved to create a blend, which I also believe to be the first implementation of the space-time blend using raster images.

The research and program can be seen as innovative as it is the first combination of these elements and in some cases new implementations and even

new methods for calculating colour in a blend between 2 heterogeneous objects, which can be expanded upon in future by improving the work and taking it further by implementing some of my suggestions and exploring alternative approaches to interpolating attributes of 2 heterogeneous objects.

References

[1] Adzhiev, V. and Pasko, P., 2004. Function-based shape modelling: mathematical framework and specialised language. *Automated Deduction in Geometry* [online]. Available from: `http://link.springer.com/chapter/10.1007/978-3-540-24616-9_9`

[2] Burtnyk, N. and Wein, M., 1976. Interactive Skeleton Techniques for Enhancing Motion Dynamics in Key Frame Animation *ACM*. Ottawa, Canada.

[3] Farneback, G., 2003. Two-frame motion estimation based on polynomial expansion. *SCIA'03 Proceedings of the 13th Scandinavian conference on Image analysis pp 363-370*. Berlin.

[4] Harrison, P., 2005. Image Texture Tools. *Monash University*, Melbourne, Victoria.

[5] Jin, B. and Geng, W., 2015. Correspondence specification learned from master frames for automatic inbetweening. *Multimed Tools Appl, 74: 4873*. [online]. Available from: `http://link.springer.com/article/10.1007/s11042-013-1847-4`

[6] Leymarie, F. and Levine, M. D., 1992. A Note on Fast Raster Scan Distance Propagation on the Discrete Rectangular Lattice. *CVGIP-IU, Vol 55(1), pp 84-94*.

[7] Mitton, R., 2009. Signed Distance Fields. *coders notes* [online]. Available from: `http://www.codersnotes.com/notes/signed-distance-fields/`

[8] Mxgmn. Wave Function Collapse. Source code available from: `https://github.com/mxgmn/WaveFunctionCollapse`

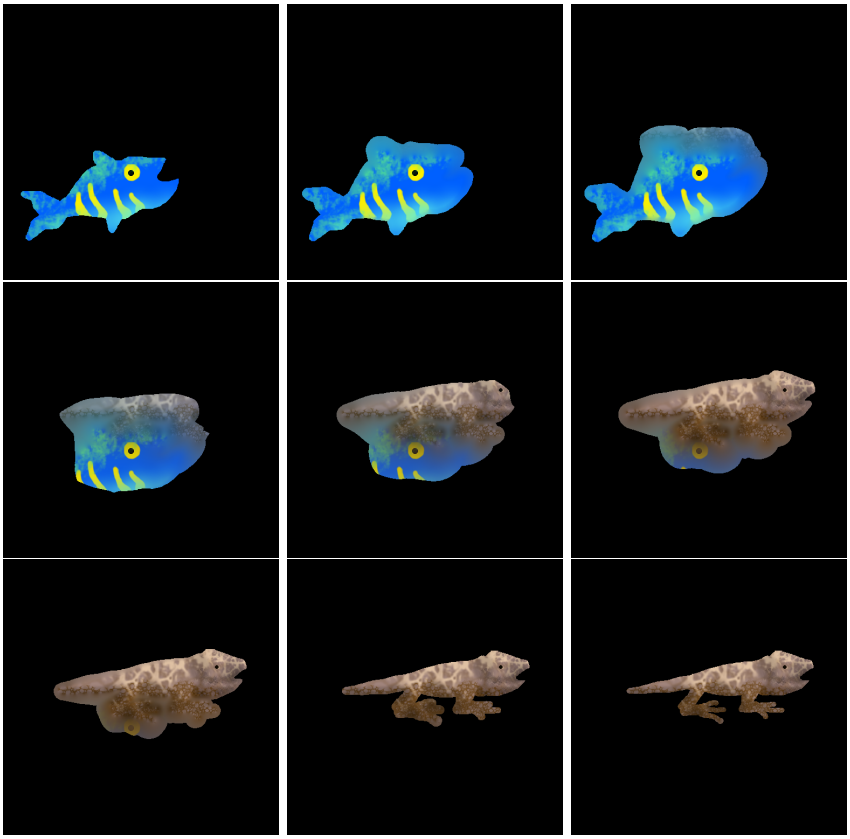
[9] Pasko, G., Pasko, A., and Kunii, T., 2004. Space-time Blending. *Computer Animation and Virtual Worlds* [online]. Available from: `http://onlinelibrary.wiley.com/doi/10.1002/cav.12/abstract`

[10] Pasko, G., Kravtsov, D., and Pasko, A., 2010. Real-Time Space-Time Blending with Improved User Control. *Motion in Games* [online]. Available from: `http://link.springer.com/chapter/10.1007/978-3-642-16958-8_15`

[11] Sanchez, M. et al., 2014. Space-Time Transfinite Interpolation of Volumetric Material Properties. *Transactions on Visualization and Computer Graphics*.

[12] Yuan, Z. et al., 2012. Object-Space Multiphase Implicit Functions. *ACM TOG 31(4).*

7 Appendix



The source code and a Linux executable can be found at `https://github.com/i7621149/SDL_spacetime_blend`.

7.1 Image Prediction Algorithm

The algorithm used for frame prediction takes 2 frames and is simple in its approach:

```
image_out = image
for each pixel in image:
        fxx = Fx(x, y)
        fyy = Fy(x, y)
```

```
        pixel = image[x][y]
        image_out[x+fxx, y+fyy] = pixel
```

There is a little more complexity in terms of comparing flow size to decide the final colour for the image, but the core algorithm remains reading the flow direction with $Fx(x, y)$ and $Fy(x, y)$, then offsetting the pixel by that amount for the final image.

7.2 8SSEDT Algorithm

For more information see [6]

```
Step 1: initialise grids as points either inside or outside
(where inside = {0,0} and empty = {INF, INF}):
For each pixel:
        If pixel is coloured:
                Grid1[x][y] = inside
                Grid2[x][y] = empty
        Else (pixel is black):
                Grid1[x][y] = empty
                Grid2[x][y] = inside

Step 2: run propagation algorithm, which checks neighbouring points
dx/dy and adding it to the current pixel to check if it is an improvement
The Compare function is defined as follows, and will update the given point:
        Compare( Point, Grid, offsetX, offsetY )
                Other = grid[x+offsetX, y+offsetY]
                Other.dx += offsetX
                Other.dy += offsetY
                If Other.dx^2 + Other.dy^2 < Point.dx^2 + Point.dy^2
                        Point = Other
For each of the 2 grids:
        Pass 1:
        For each column:
                For each x:
                        Point = Grid[x][y]
                        Compare( Point, Grid, 1, 0 )
                        Compare( Point, Grid, 0, -1 )
                        Compare( Point, Grid, -1, -1 )
                        Compare( Point, Grid, 1, -1 )
                        Grid[x][y] = Point
                For each x (traversed backwards):
                        Point = Grid[x][y]
                        Compare( Point, Grid, 1, 0 )
                        Grid[x][y] = Point
        Pass 2:
```

```
        For each column (traversed backwards):
                For each x (traversed backwards):
                        Point = Grid[x][y]
                        Compare( Point, Grid, -1, 0 )
                        Compare( Point, Grid, 0, 1 )
                        Compare( Point, Grid, -1, 1 )
                        Compare( Point, Grid, 1, 1 )
                        Grid[x][y] = Point
                For each x:
                        Point = Grid[x][y]
                        Compare( Point, Grid, 1,  y )
                        Grid[x][y] = Point

Step 3: Combine the 2 grids to produced a signed distance field
        Point1 = Grid[x][y]
        Point2 = Grid[x][y]
        Dist1 = sqrt( *$Point1.dx ^ 2 + Point1.dy ^ 2$* )
        Dist2 = sqrt( *$Point2.dx ^ 2 + Point2.dy ^ 2$* )
```

7.3 2D Shapes, Operations and Transformations

Although the final program does not support functional representation of shapes, the source code for the Shapes class, found in *Shapes.cpp*. The following shapes and transformations were implemented and used for testing:

```
Box(pos, size, radius):
        return radius - max(abs(pos) - size, 0.0).length()

Circle(pos, radius):
        return radius - pos

Line(pos, start, end, width):
        dir = start-end
        len = dir.length()
        dir /= len
        d_product = dot((start-pos), dir)
        proj = max(0.0, min(len, d_product)) * dir
        return width / 2.0 -((start - pos) - proj).length()

Translate(pos, t):
        return pos - t

Rotate(pos, angle):
        x = pos.x * cos(angle) - pos.y * sin(angle)
        y = pos.x * sin(angle) + pos.y * cos(angle)
        return vec2(x, y)
```

```
Union(a, b):
        return a + b + sqrt(a*a + b*b)

Intersect(a, b):
        return a + b + sqrt(a*a + b*b)

Subtract(a, b):
        return Intersect(a, -b)
```