

4. Designing the Tool

We have investigated many of the problems that the morphing tool will need to address, and identified possible approaches to finding solutions. The following chapter decides on what approaches the software will actually take and proposes some initial design features for the tool.

4.1. Rapid Development Requirement

One of the first things to note about the requirements is the timescale. Software development can overrun at the best of times, particularly as there are few reliable ways to estimate in advance the time and resources required to produce a software product. The Client will require a functioning product less than half way through the course of this yearlong project, and the design strategy should reflect this and build in an ample time buffer where at all possible. As a result programming will be a high priority in the early stages of the project, though this needs to be balanced with the research requirements.

The initial emphasis is on producing a morphing program, as this is the most fundamental of all the Client's wishes. User experience and interface heuristics are of a comparatively low priority. While these things are indisputably important in any interactive program, they are of little use if the program lacks core functionality. One consequence of this is that the initial releases may bear little resemblance to the final plans either visually or functionally, and may in fact exist more as a test suite than a standard application environment. The program will be evolutionary, and in keeping with the themes of this project, will itself morph as the code is refactored.

Finally, releases will be made available at regular intervals. This ensures that the Client always has the latest version, that any bugs can be addressed rapidly, shows evidence of progress, and gives the Client a sense of being more involved in the product's development. Releases will be made available through a mutually accessible website [WEBS] with email notification of any significant updates.

While I'm not rigidly following any particular software development methodology, it is worth noting that frequent releases, continual refactoring and constant client contact are major aspects to the Extreme Programming paradigm, though the level of testing will almost certainly be below that demanded by XP, mainly due to the time available.

4.2. A Framework for Representing the Morph

It may be desirable for the proposed tool to use more than one warping algorithm, or to change the warping algorithm at a later date. Building the morphing algorithm within a generalised framework will allow alterations to be made with relative ease in the future, though it will be at cost to the initial development time. While this isn't an essential part of the tool, it could benefit the project in its later stages.

There is the requirement that the framework be flexible enough to accommodate future changes with minimum refactoring, but such a framework would also require a significant amount of time to build and test. There are several morphing frameworks around, such as that used by Morphos [GOME99] and the mathematical approach described in [MILL02], though these are too involved to use in this instance. Consequently a moderately flexible framework be built-in so that new morphing algorithms can be 'plugged in' at a later date, but without overgeneralisation of the algorithms involved. Since this would require significant testing the workload will be

minimised by restricting the user interface to only perform tested operations, and only testing operations that have a potentially working concrete implementation. This allows for moderately rapid development whilst still releasing a stable program.

This proposed system for framework development still allows bugs to creep in, and spurious bugs are all the more likely when significantly refactored code is released on a frequent basis. To this end a simple error reporting facility will be built in, facilitating the transport of the exception's stack trace to the Developer for analysis. This will rapidly identify faults and allow for an immediate correction, it will also mean that faults that aren't spotted or reported by the user aren't missed. However, error reporting is not an alternative to proper testing, and where feasible every effort to ensure the software works correctly should be made before releasing it to the end user. Stack traces are not always sufficient to identify a bug, so the Client is encouraged to give feedback. The user should be notified of these otherwise covert communications.

Figure 4.1 shows a simplified overview of the proposed framework. It will work with *FeatureFrames*, which are considered to comprise of an image, a collection of features representing distortions, labelled uniquely within the *FeatureFrame*, and a set of property-value attributes, e.g. for storing file names. The distortion features will comprise of instances from a class hierarchy, and they will be organised such that only features with a common superclass with a sufficient enough concrete implementation can be added to a *FeatureFrame*'s collection, e.g. user-level features could be added but not abstract features, and features from two different techniques can not be mixed.

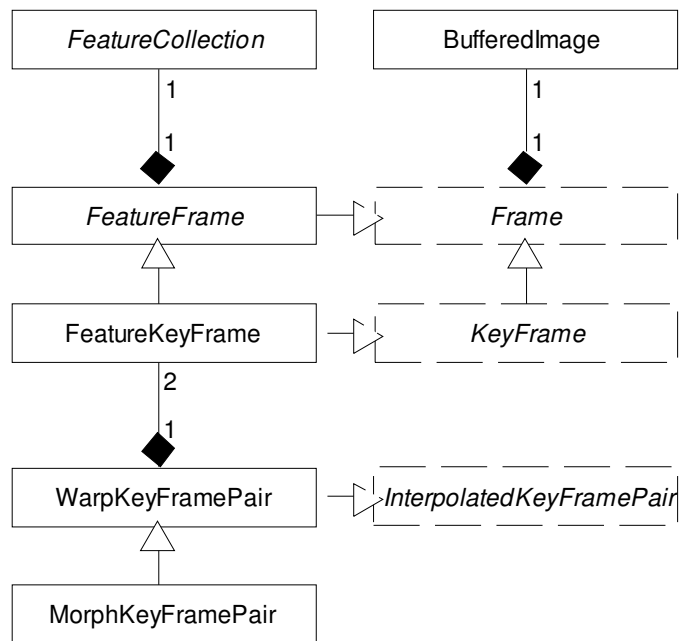


Figure 4.1, Morphing Framework. A simplified class diagram of key classes used to represent the morph.

FeatureFrames may be *FeatureKeyFrames*, two of which comprise an *InterpolatedKeyFramePair*. Subclasses of this provide facilities for generating frames between the keyframes. In the case of *WarpKeyFramePair* a rendering routine is selected based upon the nature of the distortion feature instances.

For our purposes we consider *MorphKeyFramePair* to extend *WarpKeyFramePair*. This may at first glance appear a perfectly reasonable thing to do since the two are intimately related and operate on the same images, but there are serious problems with this model. A warp takes a single image and generates a single image. A morph takes two images and calculates two warps, to which a cross-dissolve is applied. It isn't the case that the *MorphKeyFramePair* would call methods of the superclass. This approach means that one warp must be computed, then the next, meaning that the morph can't realistically be calculated progressively. While this solution isn't as elegant as we would like, it meets our needs without overcomplication.

Configuration options will be held either as properties of a `MorphKeyFramePair` or `WarpKeyFramePair`, in the first keyframe of the `InterpolatedKeyFramePair`, or in a persistent backing store.

4.3. Operating Environment

We have decided to write the program in Java 1.4. While this may result in some additional cost in rendering time, the relatively rapid development time and platform independence are helpful in meeting the Client's requirements.

The initial morphing algorithm will be Beier and Neely's field morphing technique [BEIE92] since it allows a high degree of control over the morph by letting the user pick feature contours using moderately intuitive straight lines, and the paper details the algorithm in such a way as to make implementation quick. Again, this will be to the cost of the rendering speed since the render time is proportional to the number of feature lines. Future releases may extend the straight lines into user-level splines.

[GOME99] identified three types of interface used to specify a correspondence:

- *Side-by-side interfaces* work by showing both source and target images simultaneously. This is the most common design used by morphing programs, but doesn't always fit neatly on screen and can confuse matters having to edit two spaces in order to create one.
- *Single-window interfaces* only show one image at a time and are sometimes used for warping since there is no destination image, and a good feeling for the distortion can be obtained by showing a path between source and target features. This style doesn't work so well with morphs as it is more difficult to check whether two features correspond.
- *Automatic interfaces* provide the user with a way to tell the computer how to automatically specify the morph. Few systems are fully automatic (unless they themselves generate the objects to be morphed), and most semi-automatic systems are hybrids that still allow some user-level feature control.

In initial releases we will use two side-by-side panes since correspondence is easier to specify when you can see both images, and there will not be sufficient automation to restrict interactive feature correspondence, though it is eventually hoped to be able to expand beyond this into a multiple document interface in which any number of internal windows may be opened, each representing a keyframe. Source and target frames could then be specified and features paired up automatically. As the product develops it is hoped that we will be able to progress towards a more automated interface.

While the propose multi-document interface is peripheral to the primary morphing function of the tool, it will act as an enabler to many deeper features, for example:

- The Client has expressed an interest in being able to create hybrids of more than two images. A multiple-document interface would allow an intermediate to be rendered into the workspace, which could then be used as an input into another morph, simulating a polymorph hybrid (as discussed in [LEE98]).
- Having multiple frames open at once allows comparison in feature placement and could speed up the production of morphs between existing frames.
- Having two images would no longer be a hard-coded assumption, allowing a simpler interface with operations no longer duplicated for each keyframe and no need to swap images during feature specification.
- Since the two images forming a morph may be quite unlike they may require different but corresponding templates. This would mean that features could not

be mirrored and association would be determined from some labelling system, perhaps hierarchical. This fits in better with windowed interface and would allow name and template palettes.

It may be desirable to have a hierarchical labelling system for features, e.g. pupils are part of the eyes, which are part of the face. This would allow corresponding features to be paired instantly across keyframes. This would be difficult to manage in a side-by-side layout and wouldn't allow for copying feature structures from third images. ???@ @ @ @

- @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @

Once the interface is sufficiently specified we can start to apply a more intuitive design. Nielsen [NIEL93] identified ten important heuristics for effective user interface design:

- *Simple and natural dialogue* – information should be presented in a logical order without excess complexity.
- *Speak the user's language* – use non-system terms from the user's perspective.
- *Minimise the user's memory load* – group related items, hide irrelevant features, and make use of patterns familiar to the user.
- *Consistency* – a familiarity with the interface allows rapid learning and exploration of features.
- *Feedback* – inform the user what is happening and alert them of any irreversible actions. Distinguish between passive status and active alerts.
- *Clearly marked exits* – undo and cancel should be available at a glance.
- *Shortcuts* – advanced users may benefit from keyboard shortcuts, macros, scripting interfaces etc.
- *Good error messages* – short, clear messages that don't blame the user and suggest a way to rectify the problem.
- *Prevent errors* – a good interface can stop some problems ever occurring. Modes partition functionality and can make error prevention more difficult. Where they are used the selected mode should be clearly stated.
- *Help and documentation* – searchable task-oriented help is more likely to be read by a user than manuals, and context-sensitive help can add to clutter.

The design of the interface will take these into consideration, particularly in later releases once the priority aspects of the system have been completed. In particular, the user interface will be styled to integrate into the native Operating System's look and feel, and style guidelines where this is feasible. On the Mac there are some special considerations to be taken into account, including a menu bar integrated into the top of the screen, implementation of the items in the system menu without duplication elsewhere, buttons only ever appearing a single colour, and arranging the window so that the resizing glyph isn't drawn over key components.

No satisfactory solution has been found to the issue of feature specification, though the best examples seem to strike a balance between overspecifying the mouse and providing too many states. Considering that the program may be used in an environment with only one mouse button it is important not to rely on a second or third button. We have decided that three states are needed – one for adding features, one for editing features, and a third for selecting features. By enabling features to be selected we can introduce an ability to affect groups of features, for example, features can be selected and then deleted by clicking a delete button. While we could have a delete state to do this, it would just confuse the user and require more mouse clicks. By

making deletion a two-stage process it reduces the chances of the user deleting lines by accident. To speed up selection the right mouse button will be used as a selection tool that can be invoked from any state.

Features will be displayed as straight lines. Most examples citing the Beier Neely method [BEIE92] usually show the directed line segments with arrowheads. While this can present a methodical consistency to the user (e.g. a face is represented by arrows moving clockwise around it) we deem it to be unnecessary visual clutter that could overload the user's memory. The actual direction of the vectors isn't important so long as they are in the same direction in both source images, so no distinction will be made between the start and end of a feature, but to provide the user with feedback hovering near the end of a line will ring that end. The line will be in one of several customisable colours (defaulting to green since the human eye is better able to distinguish shades of green than other colours, hence green lines should be visible on most images) depending on state – normal, highlighted (the mouse is over the line), selected and highlight-selected. An xor mode will also be available in case the lines remain difficult to see.

Feature correspondence would appear to be best addressed by mirroring the features to the second image as they are placed on the first, then allowing editing. This system has its problems – especially if features are near one another – but morph specification is tedious enough without having to manually select pairs of lines. This might not work so will in the multiple document interface as it would be unclear to which image(s) features should be mirrored, though it may be possible to set this through a frame property flag.

Help will be provided on the project website, but since the nature of the software is constantly evolving it will not be a priority to update the documentation unless there have been significant changes. While the program is under development the documentation will only provide limited help to the user.

4.4. System of Releases

The first set of releases will enable basic specification of a morph in a window with two image panes, which allows rapid development and testing of new features. Images must be loaded into either pane and the render is preset to morph between them. Renders will appear in a separate window, with each version giving more control over the render.

The second set of releases will introduce the multiple document interface. While a nested window system may initially confuse users, it adds significant power to the program and is an important stage in its evolution. Since each frame will have its own window there will be no need to have separate load and save options, and the notion of a workspace will be decoupled from the notion of a pair of keyframes forming a morph since the workspace may contain more than two frames. This will realise the client's desire for 'triangular' morphs. Feature correspondence will be more difficult since it may no longer be possible to automatically mirror features. It may be possible to create some semi-automatic pairing system.

Specifying a render would be significantly different. As the source and target frames aren't known these must be specified before anything can be generated, but this eliminates the need to be able to swap source and target images. Once an image is rendered it can itself be added to the workspace and used as a keyframe in another morph, allowing some degree of polymorphing.

Subsequent releases will build further on this more modular framework to include a hierarchical labelling system so that features with the same name are automatically paired, allowing two frames with similar configurations to be loaded in and a morph to be generated almost automatically. In addition to individual features, groups will be able to move and be resized collectively.

These releases will also examine how the domain can be exploited to allow more automatic specification of morphs. A likely starting point will be a system of templates that can be dragged and dropped from a template library. These would integrate into the hierarchical labelling system, reducing the amount of work required for correspondence. In addition, any group within the hierarchy could be exported to the template library. Later versions may then attempt to put some automation to the process.