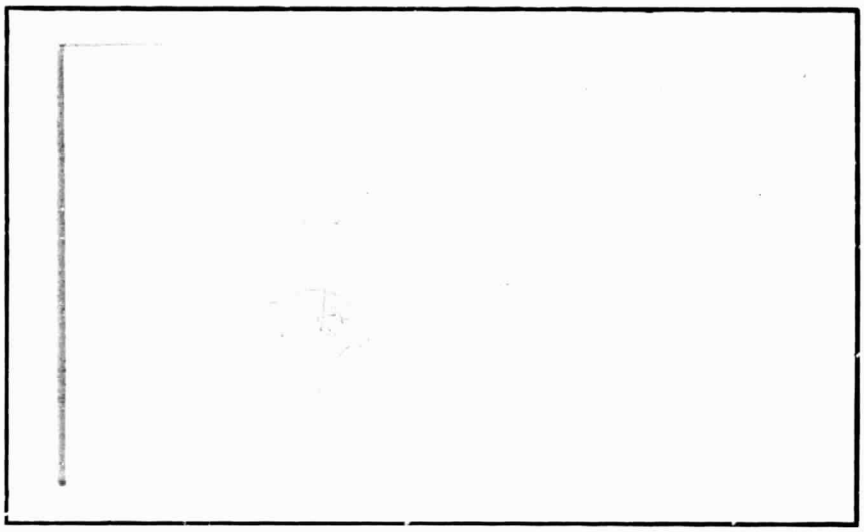

UNIVERSITY of PENNSYLVANIA
Department of Computer and Information Science
The Moore School of Electrical Engineering
PHILADELPHIA, PENNSYLVANIA 19104

System Integration Report

May 31, 1985

NASA Contract No. NAS9-16634

Norman I. Badler,
Principal Investigator

Jon Korein,
Research Specialist

Craig Meyer,
System Programmer

Kamran Manoochehri
Jane Rovins
Jeffrey Beale
Brian Barr

Department of Computer and Information Science
Moore School D2
University of Pennsylvania
Philadelphia, PA 19104

Table of Contents

List of Tables

1. Introduction

We view the long-term goal of the OSDS to be a set of application-oriented systems with a common and consistent user and graphical interface. The primary function of a good human interface is to provide access to information and capabilities without burdening the operator with the internal details of the application system. In particular, the specific databases used are not available directly to the user, but are always accessed through insulating procedures. This idea promotes system integration without requiring internal system homogeneity. PLAID, SurfsUP, and MCAUTO, for example, can co-exist through standard geometric, topological, and attribute representations. Similarly, though perhaps somewhat more easily, various sources of anthropometric, strength, and motion data may be integrated through a common intermediate representation, while each is actually obtained in a form specific to either an existing system (such as Selspot) or gathered from previously published sources (such as population statistics).

Of course, reduction of data and representational redundancy is a desirable goal, but not at the expense of tearing all systems apart to achieve commonality. We therefore perceive system integration as more of an evolutionary than revolutionary process: fitting pieces together within a carefully designed global framework with sufficient modularity to grow as the applications and tasks require.

In this report we will examine several areas that arise from the system integration issue. In the sections which follow we will discuss intersystem analysis as it relates to software development, shared databases and interfaces between TEMPUS and PLAID, shaded graphics rendering systems, object design (*BUILD*), the TEMPUS animation system, anthropometric lab integration, ongoing TEMPUS support and maintenance, and the impact of UNIX and local workstations on the OSDS environment.

2. Intersystem Analysis

Our experience with TEMPUS and SurfsUP has given us considerable insight into potential design modifications for PLAID. As noted above, these systems should evolve toward display device independence, coordinated user interfaces, and more integrated databases.

2.1. Display Device Independence

One of our strongest recommendations is to place all the graphics application software in the OSDS on top of a device-independent graphics package. While we have been developing our own CORE system for this purpose, it is becoming clear to us that the future support and extension of our CORE to new devices is becoming less attractive, both in terms of effective use of manpower and efficient support of a variety of devices here and at JSC. Either our CORE must be supported and extended, or a commercially supported system should be engaged.

Each alternative has pros and cons. The continued development of our own CORE system has the following advantages:

- We control the source code version.
- New devices can be interfaced at low cost.
- New interactive devices can be interfaced even when those devices extend the original CORE specifications (such as the six degree-of-freedom Polhemus digitizer).
- Selective implementation of CORE features, while counter to the CORE design philosophy, permits more efficient and cleaner code.
- Selective extension of CORE permits generally useful features to be added at the graphical primitive level. So far, these extensions include segment grouping to make viewport update more efficient, anti-aliased line style, back-to-front display, and depth-cueing.

The disadvantages are:

- We are (should be) responsible for maintenance of the source code.
- We are responsible for new device interfaces (though this could be performed locally at JSC if necessary).
- Some desirable CORE features may not be immediately available.
- Some non-standard extensions may be convenient to code though limiting pure portability.
- Features are added on an "as-needed" basis.
- Efficiency is not as important as reliability, ease of coding, and readability; thus the choice of PASCAL as the implementation language.

The use of a commercial software CORE package, such as DI-3000 from Precision Visuals, Inc. has the following advantages:

System Integration

- The package is supported and maintained to the standard by a commercial enterprise with facilities for handling customer inquires, bugs, and updates.
- A full CORE implementation is available.
- There is a good user manual.
- The implementation language is (in the case of DI-3000) standard FORTRAN.
- The package is available on many common computers.
- A user community for that package exists.

On the negative side:

- The efficiency of the system is open to question. Since full CORE is supported, an extremely large number of situations can arise which must always be handled in a general, device-independent fashion.
- The program structure of DI-3000 may be less well structured than one might desire in a large software package. In particular, it is reported that the device interface is not as localized and clean as it could be.
- New device drivers must be coded--based on a supplied skeleton driver-- and interfaced to DI-3000 by the customer.

We urge that the OSDS systems be converted to utilize a CORE graphics software system, such as our CORE or the DI-3000 package from Precision Visuals, Inc. At the present we favor the latter choice. There are a number of reasons for this besides the advantages cited above:

1. DI-3000 is a complete and presumably debugged CORE graphics system with a number of off-the-shelf device drivers.
2. DI-3000, like our own CORE, is a fully 3-D system unlike the other available standard, GKS. (We do not think the use of GKS is warranted at this time due to its inherent 2-D nature and the lack of consistent 3-D proposals.)
3. TEMPUS already runs on our own CORE system, so conversion to DI-3000 should take minimal effort, though some changes to functionality may be required.
4. PLAID will probably have to evolve to a more device independent form anyway as new graphics display equipment with high performance and additional features is installed or acquired at JSC. (We would rather not assume the maintenance function for PLAID systems built upon our own CORE implementation.)
5. Our current support of CORE would cease and our energies could be applied

to more fruitful body modeling, data display, OSDS modeling, and animation tasks.

At the present time the University of Pennsylvania Graphics Research Facility is contemplating the purchase of a licence for DI-3000 from another grant. In a companion lab there is an implementation of DI-3000 running on a VAX-785 under UNIX, using an Adage 3000 as the display device. In order to roughly assess the efficiency of DI-3000 against our CORE we benchmarked both on a simple task. Both systems were running on a VAX-785 under VMS version 4.1. The controlling routines were written in PASCAL. Our CORE is written in PASCAL; DI-3000 in FORTRAN. Two groups of tests were run, one to display BUBBLEWOMAN as about 2000 polygons, the other to display numerous vectors. Both experiments ran the primitives though the definition and viewing pipeline using retained segments to force display list storage of all primitives. The timings do not include any actual display device overhead. The results are summarized in Table 2-1. As may be easily seen, our CORE runs at about 90% of DI-3000 speed on polygons, and about 70% of DI-3000 for lines. Thus conversion to DI-3000 would represent an increase in speed of between 10%-30%, a savings we consider very significant.

Table 2-1: DI-3000 / CORE Comparison

Time is in milliseconds.
Percentage is = DI-3000/CORE.

TEST	CORE(ms)	DI3000(ms)	Difference(ms)	Percent
Polygons				
1 woman	78840	70470	8370	89.4%
2 women	156580	141390	15190	90.3%
			Average	89.85%
Lines				
2000	11980	8350	3630	69.7%
4000	23310	16480	6830	70.7%
8000	47670	32390	15280	67.9%
16000	94860	64770	30090	68.3%
			Average	69.15%

Any TEMPUS conversion to DI-3000 will require us to write a DI-3000 driver for

the Grinnell display and convert the graphics calls in TEMPUS to the DI-3000 routine names. There will be a slight feature mismatch as noted, however, since:

- our CORE may have certain characteristics which vary from the strict CORE standard proposal, such as the way we handle multiple viewports, the grouping of segments for viewport erasure efficiency, the back-to-front priority display, and anti-aliased line drawing;
- DI-3000 will have features we were unable to take advantage of in our CORE, such as fully asynchronous input and device echoing;
- DI-3000 is written in FORTRAN while our CORE is written in PASCAL, requiring some changes in the present parameter structure of the calling routines.

We do not anticipate major problems, however, since such conversions or adaptations are what standards are designed to facilitate. The conversion task would probably take several months, not only to convert code but to take advantage of DI-3000. If NASA JSC approves the conversion to DI-3000 we will continue to support the CORE version until the conversion is deemed satisfactory.

As a potentially interesting aside, we will note that there is now an IBM PC implementation of DI-3000. Running on a separate, on-board processor, this system completely off-loads any graphics memory overhead from the host PC. All DI-3000 device drivers can be supported, making the PC into a true graphics application host. Although neither TEMPUS nor PLAID will yet fit into a PC, it is useful to note that the standardization process has finally touched on a very popular and capable system. We have been unsuccessful in fitting our own CORE onto a PC for two reasons: there is no PASCAL compiler that accepts the large number of procedures in CORE and produces object libraries, and even if it did, these procedures would use up the processor's main memory and squeeze out most of the application. The latter problem is somewhat alleviated in the larger IBM-XT and -AT, but the former problem remains.

2.2. Coordinated User Interfaces

The recommended DI-3000 system also includes a much better (or at least cleaner) implementation of the user input system. By shifting inputs to DI-3000, both PLAID and TEMPUS would assume a common interactive device interface mechanism. With the proliferation of input tools (tablets, mice, keyboards, dials, *etc.*), maintaining clean interfaces and common functionality is an increasing chore for our group. The OSDS environment should aim for transportability of input devices between displays and

systems to maximize their usefulness.

One possible problem with this arrangement is caused by the extension to CORE we recently added for the Polhemus 6-axis digitizer. CORE (and GKS is certainly no better on this matter) only supports input devices up to 3-D. The extension to 6-D is straightforward, but should be made in order to fully utilize the Polhemus. It is possible that future standards will be formulated to support the 6-D system; if not, the Polhemus is unique enough that a separate subsystem for it can be accessed through the (standard) *escape* mechanism.

Another area in which the user interface may be made more consistent is in the standardization of input selection. Currently PLAID uses at least two kinds of command inputs (strings and menu picks (in BUILD) from the keyboard), and locator (crosshair) inputs for picking and coordinate selection. TEMPUS uses menu selection either from the keyboard or from the graphics screen via a tablet- or mouse-based locator and pick. Near future TEMPUS capabilities will also include valuators and locators based on the Polhemus digitizer. The ideal situation would be to place TEMPUS, PLAID, and especially the oft-discussed new BUILD under the same, consistent, user interface. The process would probably involve changing both PLAID and TEMPUS interfaces somewhat: TEMPUS would be extended to include a keyboard command mode which bypassed menu selection (unless it was specifically requested in a *learning, novice,* or *help* mode); PLAID would be converted to use the same interactive front end as TEMPUS. The new BUILD could be constructed directly with existing TEMPUS interactive routines, augmented by geometric positioning as presently exists or as will exist when the combination of a real-time display with the real-time 6-axis control becomes a reality.

In the near future we anticipate that TEMPUS will depend on only two non-keyboard devices for input. One is the mouse which will replace the eternally flaky tablets. The tablet has not been as useful for TEMPUS as we originally imagined. The new generation mice are inexpensive, "intelligent" enough to be connected directly to the alphanumeric terminal, and do not require their own serial communications port to the host computer. At $150 or so they are nearly disposable if they malfunction. The second device is the Polhemus digitizer. At $20,000 this would have been an expensive proposition, but now a $3,000 version, called the *Isotrak*, has been announced. At this cost, it is about the same as a large digitizing tablet and far more versatile. Although its

accuracy is about 5mm, this is quite sufficient for pointing and picking operations.

Given the finite resources available for software engineering, the conversion of the input systems to a consistent user view interface is somewhat lower priority than the integration of the graphical input and output systems.

2.3. Integrated Databases

As part of the current contract we were to examine the possibility of providing a common geometric database for TEMPUS and PLAID. The candidate sytems were few, and we considered IGES (*I*nitial *G*raphics *E*xchange *S*pecification) the primary contender [6]. After beginning to implement an IGES interface between TEMPUS (SurfsUP) and PLAID, the project was shelved for lack of standards on topological structures. IGES does not yet officially define the format of object topology: the vertex, edge, face, solid (*csurf*), and collection of solids (*psurf*) structures that pervade TEMPUS and many other modeling systems, for that matter. Lacking this specification, we were forced to abandon the IGES standard *for now* as the medium of communication between PLAID, TEMPUS, and other CAD/CAM systems such as MCAUTO. There are some efforts to remedy this situation, but they are in a very young stage of development [9].

We have implemented a two-stage translator from PLAID *COG* and *primitive files* to *target* files, and then into SurfsUP *psurf* files. The inverse transformation from *psurfs* to *COG* files also exists. This code should be integrated better into both TEMPUS and PLAID, so that external processes need not be run to convert databases. The status of this integration and current outstanding problems are reviewed in the next section.

3. The PLAID-TEMPUS Interface

Our goal in this section is to outline the steps necessary to completely interface PLAID and TEMPUS objects. It is not suggested that every indicated task be implemented immediately. It is prudent to prioritize the most essential needs first. Each item will include a brief discussion of its importance and the anticipated difficulty of the implementation.

It is convenient to divide the discussion into four sections. The first reviews the current state of the code. The second outlines what will be necessary to allow full PLAID

input to TEMPUS. The third discusses TEMPUS output to PLAID. Finally, the fourth summarizes how recent and future changes to PLAID may be accomodated.

3.1. State of Current Interface

COG objects can be read into the TEMPUS *psurf* hierarchy for use by all positioning and line-drawing display commands. Primitive files must be preprocessed into target files, and all files for a given COG object must be in the same directory. A utility program has been created for easily turning primitive files into target files from the VMS command level. The main limitation is the inability to use either the TEMPUS *shaded* system or clearance commands on COG objects with holes or bow-ties. The solutions to these problems are addressed in Section 3.2 below.

Currently, TEMPUS is able to output a body as a PLAID primitive. This gives users the ability to position a body in TEMPUS, and then output the position to PLAID. The body position may then be modified globally in PLAID and used for picture output, but the joint angles are not adjustable. Extensions to achieve full TEMPUS output to PLAID are discussed in Section 3.3.

3.2. TEMPUS Input: Clipping and Object Decomposition

Two tasks are necessary for TEMPUS to accept all COG objects as input to all TEMPUS facilities. These are the clipping of objects for shaded output, and the decomposition of bow-tied objects and objects with holes for both the *shaded* system and the clearance routines. The latter may also be necessary in the future for other TEMPUS procedures, as it is difficult to even determine a surface normal for such objects. If TEMPUS is to use CORE's ever-improving polygon handling capabilities, decomposition routines are essential.

Clipping must be done in two places. The first is in the polygon preparation routine of the existing *shaded* code, which is used to prepare polygon output for the Lexidata. The second is in Crow's rendering code (see Section 4), which currently lacks back and front clipping planes. Both can be interfaced to the Sutherland-Hodgeman clipper coded for our CORE system. The interface tasks will be of moderate difficulty, since some modifications to the way the routines handle polygons will have to be made. They are a high priority for both systems, since COG objects must often be clipped in NASA task analyses.

System Integration

The simplest way to handle the decomposition process will be to interface to Ted Kell's *decompose* routine. This routine is used by PLAID to create convex polygons for input to the Lexidata, but it looks likely that we can adapt it to our purposes. The main interface problem is that Kell's routine works in the x-y plane, after the viewing transformations have been applied. There are many advantages to storing the decomposed polygons, possibly as an alternate data structure in their local coordinate systems. TEMPUS has no structural modification capabilities, so the decomposition process need only be done once. In addition, certain TEMPUS procedures, such as the clearance routine, depend on a three-dimensional, unprojected, object coordinate space.

For these routines, we cannot arbitrarily project polygons onto a plane as we need an accurate decomposition in world space. Polygons whose normals are perpendicular to the direction of projection will project as lines and therefore not be properly decomposed. It is possible, however, to rotate all polygons so that their normals are in the positive z direction, and then decompose their projections on the x-y plane. Afterwards the inverse transformation can be applied to bring the decomposed polygons back into local coordinates. The result will be the decomposed polygon in its local coordinate space. There will be some tricks to doing this process correctly and making sure the decomposed polygons are oriented correctly, but they should not be too difficult. In fact, this process strongly resembles the transformations that have already been coded to map polygons onto textures.

The decompose routine must also be given a shell to allow an entire *psurf* to be decomposed as a unit. Maintaining structural integrity should not be a major problem, but it must be done carefully, since we are now dealing with polyhedral objects. It is very likely that portions of Kell's code which eliminate collinear vertices will have to be changed, since all original vertices must be maintained for TEMPUS purposes.

It is recommended that decomposed *psurfs* be calculated the first time they are need and stored internally afterward if they differ from the original *psurf*. If the decomposition routine is slow enough, it may be worth storing them in a file as well. In general, speed will win most trade-offs with storage.

The task thus may be reduced to the writing of the decompose *psurf* routine, including the interface to Kell's code, the auxilliary routines necessary for storage and retrieval, and the usual integration with existing code, in this case the shading and

clearance routines. The project is fairly large, but also of very high priority.

3.3. TEMPUS Output to PLAID

Two modifications are necessary to allow full output to PLAID from TEMPUS. The first addition required is a command which will output a workstation part as a COG object. This will allow an entire TEMPUS scene to be given to PLAID for display or analysis purposes. While the workstation object and body global position would still be adjustable, body joint angles will still be inaccessible. A procedure which does this workstation output has been coded, but still has some bugs. Fixing it will be a minor task.

The second change necessary for complete PLAID compatibility is the output of TEMPUS bodies as COG objects, rather than primitives. This will give the PLAID user the ability to adjust joint angles of scenes created in TEMPUS. Some of the low level code used for writing out workstation part COG objects can be used for this, but a new top level routine must be written to call these procedures while traversing the body structure appropriately. Writing this routine will be a medium level project.

Both tasks are of medium priority. It will be very nice to have TEMPUS be able to output to PLAID, even though the usual flow of data is from PLAID to TEMPUS.

3.4. Recent and Future PLAID Modifications

PLAID, like TEMPUS, is undergoing constant revision, usually with the goal of creating a more powerful system. It is important for TEMPUS to use this power as much as possible. This section briefly discusses what PLAID changes are occurring which TEMPUS will need to accomodate in the future.

PLAID has recently been changed to use a fairly sophisticated system for retrieving file names. A detached process, which we shall call the *task manager*, uses predefined tables to look up the appropriate directories in which PLAID users' files reside. This process essentially takes a simple filename as input and returns a fully specified (directory, *etc.*) filename as output. For full compatibility, both TEMPUS and our preprocessing routines, which convert primitive files to target files (the "targeting process"), should derive PLAID file specifications through the task manager. We would recommend that this change also automate the targeting process so it would be transparent to the user. This would require integrating it with the input COG command.

Taken together, this set of tasks is a fairly large project. It is of moderate priority right now, but will likely become more important as TEMPUS is used with a greater diversity of PLAID objects in varying places, and the inconvenience of copying all PLAID objects into one directory becomes greater. The change should not begin, however, until PLAID is running under VMS version 4.0 and we are able to get a version of the task manager running at Penn.

Some simpler, but equally important, changes to PLAID are planned in the near future. These involve storing topological and color information about COG objects in the COG record. The structural information will indicate whether an object is closed, mostly closed, or open. This information can be used by various procedures; the shading system, for example, should treat backfacing polygons differently for closed and open objects. The color information is stored as symbolic indexes into a predefined color table, which will store the HLS values for the color. Clearly, this information can be easily retrieved by TEMPUS. Neither of these tasks are difficult, and both are medium priority.

3.5. Recommendations

Our own assessments aside, it is important for users at JSC to look at these tasks and prioritize them for us, recognizing that the ones that are put on top of the list are the ones that will be completed sooner. Nonetheless, we shall make some tentative suggestions. The changes needed for complete use of PLAID input, the clipping and decomposition routines, are probably the most important. If there is a need for TEMPUS output to PLAID, completion of these routines is a high priority; otherwise, adapting to new PLAID features is more important.

In the longer term, the database for both PLAID and TEMPUS could be the same. The features required of both could be merged into a single unified format. Both systems require object structure as a *hierarchic polygonal boundary representation*, a fact which leads to major commonality of form and purpose. The major TEMPUS features required beyond those in PLAID are the graphics-related attributes of *color, texture, translucency, material, etc.* TEMPUS and PLAID differ slightly on how smooth edges are represented: PLAID has *invisible edges*; SurfsUP requires edges to be specifically flagged as smooth transitions. SurfsUP graphics will also support the implicit definition of a smooth edge if its dihedral angle is less than some given threshold.

Upward compatibility of existing PLAID databases with any new formats is an important issue and should be addressed with care. JSC should assess whether one-time conversion of existing databases is justified in terms of future compatibility with a common TEMPUS-PLAID format. It is not too difficult to write conversion programs for a small number of existing formats and convert data as required.

One advantage to hard conversion is that errors and poor design characteristics may be cleaned out. For example, it would be most useful to have geometric and topological data conform to certain structural standards: consistent polygon traversal (for consistent outward-facing normals), deletion of dangling edges, enforcement of polygon planarity, insertion of intersection vertices, removal of degenerate (effectively zero-area) polygons, *etc.* We understand that some effort to do this to the existing PLAID database is already underway. Some of these problems may be tested algorithmically by procedures already in place in SurfsUP (and possibly in PLAID). It should be noted that raster (shaded) graphics displays and especially shadowing algorithms are very sensitive to adequate input data.

Common database formats for *lights* and their attributes, *cameras* and their viewing parameters, and *people* must also be agreed upon. Given the role of TEMPUS to define and manipulate these objects, the TEMPUS formats can be adopted by PLAID as needed. The format of this information, as well as the hierarchical polyhedral models, has not benefitted from any substantial standardization effort. The IGES standard does contain attributes which are specifiable by the application program, but it does not appear to be useful to adopt only part of a standard when the important parts (for topology) have not yet been stabilized. Moreover, the IGES format is extremely costly in storage space even for relatively simple objects [9] and may be nearly impractical for the objects necessary to the NASA group. Other attempts to incorporate topological information into IGES apparently are underway [7].

One alternative to IGES might be found in the formats for existing shaded graphics rendering systems. Some inquiries into the most capable and respected rendering systems ever written have pointed to two systems: MOVIE.BYU from Brigham Young University and a system written by Frank Crow when he was at Ohio State University. The input formats for these systems have been used in several other systems, and may eventually be adopted for certain hardware systems as *de facto* standards.

Since human figures are fundamental to both PLAID and TEMPUS systems, a representation both as COG and primitive objects and as a parameterized entity should be supported. In the latter case, sufficient information is stored to enable the re-creation of a specific body (and its suit) by name, statistics, or anthropometric parameters.

One strong recommendation which we must make here is the coordination of design efforts between JSC and its contractors, including our group, to insure that re-design of any of the various OSDS systems be reviewed by other concerned groups. We should be asked to review any design specifications for contemplated PLAID database changes; in turn we expect feedback from JSC and its staff and contractors on the impact of our system design on their efforts. This coordination is essential to effective database exchange and system integration.

4. Shaded Graphics

Our experiences with the development of the TEMPUS *shaded* graphics system have given us an interesting view of graphical capability engineering. We have come to appreciate the quantum leap necessary to turn good ideas into production software. The difficulty with developing a shaded graphics rendering system is not in the conceptual stages, but in the detail and efficiency coding that makes a system reliable, effective, and, of course, fast.

Our present *shaded* graphics systems meets the NASA contract requirements in a formal, but not really practical, fashion. To its advantage, the *shaded* system:

- Produces solid renderings of SurfsUP polyhedra or PLAID COG and primitive target files,
- Interfaces to either a "standard" frame buffer (like the Grinnell) or a Lexidata Solidview system using its inherent z-buffer capability,
- Uses a simple surface reflectance model,
- Allows arbitrary finite or infinitely distant multiple light sources,
- Allows a camera position based on CORE viewing parameters.

On the more negative side, several features are less useful or stable:

- Full polygon clipping, so that images of object interiors can be produced, will be available soon.
- Texture mapping facilities are not quite incorporated, though arbitrary

textures may be generated and positioned onto polygons using IntSurf functions.

- The shadow capability exists in a very tenuous form: the code to find silhouettes is restrictive and in need of further debugging. The algorithm itself, while novel and published, is too massively inefficient in practice.

- The rendering code is too tightly interwoven into the SurfsUP modeling system; there should be a clean separation, via a file structure if necessary, between objects generated for display and the display algorithm itself.

- The method of anti-aliasing by supersampling is acceptible, but not particularly efficient.

These observations have led to the conclusion that we should search for available source code rendering systems. There are two candidates that we have examined: MOVIE.BYU and Frank Crow's system. The MOVIE.BYU system is a FORTRAN source rendering system available for $1000. We have a version of it at the University of Pennsylvania and have even used it to produce shaded images before our own *shaded* was running. There are well over a thousand installations of MOVIE.BYU and therefore a substantial user community with its own User Group. Among the advantages of MOVIE.BYU are

- Low cost.

- Wide distribution, including versions on all major computers and supercomputers.

- FORTRAN language based.

- Yearly updates.

- Anti-aliasing.

- Multiple light sources.

- Shadows.

- Translucency.

- Good user manual.

- Continuing development.

The primary disadvantages to MOVIE.BYU are

- Fixed, inflexible input format.

- Unusual viewing parameters: all objects are assumed defined relative to a

fixed origin and viewed down the z-axis; thus the object space must be translated and rotated rather than the camera. Although awkward, the CORE viewing parameters can be transformed into the correct MOVIE.BYU parameters by straightforward matrix operations.

- Object storage size limits: these are due to FORTRAN array declarations and limitations in marking unused polygons on a scan line by using a bit mask.
- No texture mapping facility (until at least 1986, according to the MOVIE.BYU staff).

There is a reasonable expectation that only the last of these restrictions will be removed in (near) future versions of MOVIE.BYU. The development group supporting it at Brigham Young University has been expanded recently but they are not saying what their priorities are. We continue to watch progress in this software system, but are unwilling to perform extensive modifications to it on our own at this time.

The second system is a public domain shaded graphics renderer written in C under UNIX by Frank Crow while he was working on an NSF grant at Ohio State University [4]. Crow now distributes the code without charge. This code is now running on a VAX-785 in a companion lab. Output is produced for an Adage 3000 display or a generic frame buffer. In the latter case, the images are packed into a run-length encoded format and transferred to our VMS VAX-785 where they are unpacked and displayed on the Grinnell. To our knowledge (and based on comments from other computer graphics experts) Crow's system is probably the most complete, tested, and capable renderer available for polyhedral objects. The data file formats used by this system are quite similar to those of SurfsUP; moreover the Crow format may become a *de facto* standard interface for polyhedral model rendering systems. At least one workstation manufacturer is considering supporting this format in its own software. The advantages to Crow's system are

- Public domain code.
- Flexible input language.
- Unlimited database size.
- Multiple light sources.
- Anti-aliasing.
- Texture mapping.
- Translucency.

The disadvantages to Crow's system are

- Not supported.
- Written in C for UNIX.
- No shadows.
- Small user community

We recognize that the "C on UNIX" characteristic may not, in fact, be a disadvantage. For one thing, it permits the advantageous use of UNIX software tools to parse the input language; for another, it permits computing graphics renderings on (distributed) systems running UNIX.

We intend to proceed with future rendering systems in a three-pronged fashion:

1. Continue development of our *shaded* system to the extent that it supports the Solidview interface and polygon clipping for that and similar z-buffer hardware display systems.

2. Maintain a MOVIE.BYU interface so that this system can be called upon to produce images where there are relatively small numbers of polygons, and shadows and anti-aliasing are important (but not textures). Note that the input format restrictions require that we communicate with this system through an ASCII, readable file. This is not a disadvantage, since it means that the interface between TEMPUS and the renderer is forced to be "clean." This interface code will not be difficult to write since similar work has already been done for Crow's system.

3. Maintain a Crow system interface so that this system can be called upon to produce images where there are relatively large numbers of polygons, and where textures and anti-aliasing are important (but not shadows). The required conversions from TEMPUS psurfs to Crow objects have been done (see the next Section). The Crow system will enable UNIX-type workstations to crunch away at shaded image generation, off-loading the host VMS VAX.

Thus the system or the user can select an appropriate rendering system to optimize efficiency, image features, and available auxilliary computers. In the medium- to long-term development of TEMPUS we expect readily available (commercial or public-domain) systems to meet the projected shaded graphics needs of TEMPUS.

4.1. Input to Crow's System

Crow's system needs three types of files to produce a shaded image. First, it needs a scene description file (.scn), which basically tells which objects are to be included in the scene, where there are to be placed, where are we to be viewing the scene from, and various other features of the scene. An example of the format of this file follows:

```
call branch.obj by world
call branch.obj by all_vst_p
attach all_vst_p to world at 0 0 0
call branch.obj by b_relbow
attach b_relbow to all_vst_p at 0 0 0
rotate b_relbow about 0 0 0 0 0 1 by    24.999866696 then
        about 0 0 0 1 0 0 by    29.999680071 then
        about 0 0 0 0 1 0 by   -44.999280162
call p_larm.obj by p_larm
attach p_larm to b_relbow at 0 0 0
call branch.obj by b_rwrist
attach b_rwrist to b_relbow at 0 0 0
place b_rwrist at     0.000000000   -27.304999590     0.000000000
call branch.obj by b_fingers
attach b_fingers to b_rwrist at 0 0 0
call p_wjnt.obj by p_wjnt
attach p_wjnt to b_rwrist at 0 0 0
call p_ejnt.obj by p_ejnt
attach p_ejnt to b_relbow at 0 0 0
place light at  -100.000000000   100.000000000  -150.000000000
paint light with 1.00000 1.00000 1.00000
scale light by 4.9104e+11
place center_of_interest at     0.000000000     0.000000000   100.000000000
place eyepoint at     0.000043711     0.000000000   500.000000000
set view_angle to   0.088
render_on bb 32 1
```

The next type of file needed is an object detail file (.std), which defines all of the vertices and faces of a particular object in the scene. An example of a detail file follows:

```
data    26    34
   -4.058462850     4.058462850     0.000000000
   -5.741365189     0.000000000     0.000000000
    0.000000000     5.741365189     0.000000000
   -2.870682595     4.058462850    -2.870682595
                        .
                        .
   -2.870682595     4.058462850     2.870682595
    0.000000000    -4.058462850     4.058462850
    0.000000000    -4.058462850    -4.058462850
    2.870682595    -4.058462850    -2.870682595
    8     2     1     3     9    19    22    14     6
    4     2     1     4     7
    4     5     8     2     1
    8    19     9     3     1     2     6    14    22
                        .
                        .
    4    21    13     9    19
    4    19    22    23    20
    4    26    21    19    22
```

The last file type necessary for Crow's system is an object description file (.obj), which describes which file contains the object's detail, default attributes of the object, and which display routine will be used to render this object. An object description file appears as follows:

```
title           P_WJNT
display         poly_zsort
detail          ppwjnt.det
type            polygon
bounding_box      -5.741365189     5.741365189    -5.741365189
     5.741365189    -5.741365189     5.741365189
color           0.72000 0.75200 0.80000
shininess         1.00000
transmittance   0.00000 0.5
```

4.2. Object Data Structures in TEMPUS

The objects in TEMPUS are arranged in a hierarchical data structure. Object transformations and attributes (color, gloss, etc.) are inherited from an object's parent in the hierarchy. To compute a particular object's attribute value, traverse the hierarchy upward until some object in that path has that particular attribute. To compute a particular object's transformation, begin at the root object and traverse the hierarchy downward composing all transformations as you go along. This basically means that if a parent object moves two units to the right, then the child object moves two units to the right plus any transformations which apply locally to the child object.

There are three types of objects in the TEMPUS hierarchy; branches, transformations, and polygonal surfaces. Branches provide a means of widening the hierarchical tree, and giving a common name and common attributes to all objects beneath them. Transformations are used so that all objects under them are translated, rotated, or scaled according to the transfomation matrix stored with this object type. Polygonal surfaces are just collections of vertices, edges, and faces which describe real objects in a scene.

4.3. Interfacing TEMPUS and Crow's System

Because all of the data structures must be retrieved from TEMPUS, the interface procedures are naturally included in TEMPUS. There is a command that is part of the system facilities menu in TEMPUS, which calls the interface procedure. This procedure works as follows:

1. Open and initialize the scene description file.
2. Then, for each visible object in the object hierarchy:
   - If the object is a transformation, just compose it with transformations which have been passed down the hierarchy.

- If the object is a branch:

  a. Write to the scene file a line in the format:

     ```
     call branch.obj by <branch-name>

     {branch.obj is a null object and is
      only used to preserve the hierarchy}
     ```

  b. If the branch has a parent, write a line to the scene file in the format:

     ```
     attach <branch-name> to <parent-name> at 0 0 0
     ```

  c. Write to the scene file any local translations or rotations for this branch. A transformation is local when an object's parent is a transformation object.

- If the object is a polygonal surface:

  a. Write to the scene file a command in the format:

     ```
     call <object-name>.obj by <object-name>
     ```

  b. If the polygonal surface has a parent, write the command to the scene file:

     ```
     attach <object-name> to <parent-name> at 0 0 0
     ```

  c. Write to the scene file any local translations or rotations, and any scaling which has been inherited from all parent objects in the hierarchy.

  d. Open a picture detail file and write out all vertices and faces of this object.

  e. Open an object description file and write out the object's default color, transmittance, shininess, bounding box, and display routine.

3. Output all light source information to the scene file, including their locations and colors.

4. Output all of the viewing parameters to the scene file, such as center of interest, eye or camera position, and view angle, which defines the window the camera looks through.

5. Close the scene file.

The interface procedure has produced files which can be read and understood by Crow's system. Presently the interface procedure only outputs workstation parts but not any persons. This capability could easily be included by adding a procedure call in

WriteCrowObj, which would output persons. This would give the system the capability to render any scene which is created in TEMPUS.

Another improvement to the interface procedure would be the writing of vertex and polygon color files. These additions would be placed in *PutCrowPsurf*. Since in TEMPUS, a vertex or face can have a color which is different from the object's default color, this capability will also be a necessary addition to obtain more accurate results. One drawback of Crow's system is that only the color and transmittance attributes can be attached to vertices and faces. In TEMPUS, attributes of all types can be attached as far down in the data structure as the vertex and face levels. Therefore, if a particular face, for example, has a glossiness which varies from the object's default glossiness, this information cannot be input to Crow's system without some major revisions to his program.

4.4. Crow's Scene File Commands

Since the Crow system format is quite readable and flexible, we present its syntax here.

- **call <filename> by <object-name>**
  read in object description from file <filename> and call it by <object-name>.

- **attach <object-name1> to <object-name2> at <point>**
  set up object hierarchy with object-1 the child of object-2.

- **paint <object-name> with <color>**
  give an object a color (color = red green blue).

- **place <object-name> at <point>**
  translate object by the vector given in point.

- **rotate <object-name> about <axis1> by <theta1> [[then about <axis2> by <theta2>]...]**
  rotate object about an arbitrary axis by theta degrees. Up to 4 different rotations allowed.

- **scale <object-name> by <point>**
  scale an object in the x, y, and z directions specified by point.

- **render_on <device> [<frame-number>]**
  generate a full-quality image compatible with a specified device.

Detail File Format

data <# of vertices> <# of faces>
$x_{vert\ \#1}$ $y_{vert\ \#1}$ $z_{vert\ \#1}$
$x_{vert\ \#2}$ $y_{vert\ \#2}$ $z_{vert\ \#2}$
etc.
<# of vertices of face #1> <vert #1 of face #1> <vert #2 of face #1> . . .
<# of vertices of face #2> <vert #1 of face #2> <vert #2 of face #2> . . .
etc.

Object File Format

title <object-title>
display <display routine used to render this object>
detail <detail filename>
type <object type (usually "polygon")>
color <red green blue>
shininess <gloss exponent>
transmittance <value> <rolloff>
bounding_box <xmin> <xmax> <ymin> <ymax> <zmin> <zmax>

5. TAN: The TEMPUS Animator

The TEMPUS animation system, *TAN*, has been undergoing an evolution in its specifications over the last two years. The goal is to have a graphical interactive system with which to manipulate any TEMPUS entity, including people, workstation parts, cameras, and lights. TAN will give the user very powerful and flexible control over the timing of any of these motions. For people, the motions include positions, reaches, and empirical (*e.g.*, from motion analysis) or computed (*e.g.*, spline curve interpolation) sequences.

The organization of TAN is based on the following concepts:

Score — The complete set of information in an animated sequence. The score contains tracks, actions, variables, keyframes, dynamics, phrasing, and any other associated structures.

Track — Tracks are vertical lines which appear on the display screen along which time increases, beginning with time zero at the top. Tracks represent different things at different levels of display. At the object level, each track generally represents a TEMPUS object, though there are a set of non-object tracks for other aspects of the animation score. TEMPUS objects may be people, cameras, light sources, or 'workstation objects,' our name for arbitrary parts of the polyhedral object hierarchy. Non-object tracks include the captions and title track, the sound track, a background track, and a track simply for holding global events that are not associated with any specific objects. At the detail level, tracks are most often associated with the set of

individual degrees of freedom of TEMPUS objects. These tracks will be displayed to the right of the object level track on which they elaborate.

Event — Defines a single thing that happens at an instant of time. An event corresponds closely to a single TEMPUS command. At the most detailed level it corresponds to that part of the command affecting the variable a track represents. For example, the command *Reach x y z* on a top level person track is a single event at that level. It does, however, represent three events, one for each of the *x*, *y* and *z* variables, at the detail level. All events will be represented as short horizontal tick marks along the track at the time the event is to occur.

Variable — A motion variable. The changeable aspects of an object. Each variable reflects only one value: for instance, the translation of an object is determined by three (X, Y and Z) variables.

Dynamics — The pacing of motions. The relative timing of a sequence of keyframes determines the speed at which actions occur. Thus adjusting these timings, and thus their interpolation functions, will control the attacks, decays, accelerations, *etc.* of an animated sequence.

Keyframe Set — A set of variables, over a specified period of time, having the same time to keyframe function. The dynamics of these variables may be adjusted simultaneously by adjusting this common time to keyframe function.

Keyframe — The value of the set of variables of a Keyframe Set at a specific keyframe number.

Span — A sequence of events applying to a single track within a specified time span and all occurences between included events. For example, the interpolation functions defined between events on variable tracks are included in the span.

Action — An arbitrary set of spans and events.

Group — A named, parameterized action. This enables complex animation sequences, once constructed, to be manipulated simply as a unit. In addition to copy, move, and save, the top-level dynamics of a group can be manipulated by adjusting the dynamics of a group as a whole. Parameterization is required so that a group can be applyed to various objects at various times.

Phrasing — The smooth continuation of one variable's value in one action into the same variable's value in an immediately following action.

The current plan is to complete the specifications of TAN, then design a simple command driven system that will take TEMPUS macro files as input, interpolate the

inbetweens and output the results in a TEMPUS macro format to be displayed through TEMPUS. We are also investigating the implementation of TAN in ROSS, an "object-oriented" programming system written in LISP. The graphical components of TAN will still use PASCAL and our CORE system.

Another open question is whether to implement TAN on the VAX or on a separate, but communicating, co-processor such as a graphics workstation. There is clearly an advantage to the latter, as the user may be creating objects and positions in TEMPUS while simultaneously manipulating the database for timing and synchronization control. Real-time motion playback and editing is then possible without exiting TEMPUS [1]. We currently lean toward this approach, though it means TAN may have to be implemented in C on a UNIX system (see Section 9). This is the environment for the high-speed graphics workstations such as the IMI 500 or the Silicon Graphics IRIS 2400 we are considering.

6. BUILD

The use of a standard database interchange system (perhaps a future version of IGES or a locally supported "standard") will also significantly aid the design process when existing CAD models are available from other NASA or contracted sources. At the extreme, with body models generated through TEMPUS and workstations available from CAD systems, the burden on BUILD may be significantly reduced or at least become oriented toward modification of existing workstations.

Any contemplated re-design of BUILD should address the common input and output systems mentioned above, as well as the common database issue. At this time there is no *graphically* interactive analog to BUILD in TEMPUS. SurfsUP supports only a string and textual command system (IntSurf) for testing all TEMPUS routines prior to their integration into the TEMPUS interactive system.

Useful extensions to the existing BUILD would include both new user actions as well as various built-in system checks:

1. User specification of attributes such as color, translucency, material *etc.*

2. User specification of shading method to be applied along an edge: smooth or discontinuous.

3. User specification of texture maps for complex, essentially flat structures (such as panels), and the association and orientation of the maps with respect

to the required polygons.

4. System checking to insure that all polygons are complete, planar, closed, and consistently oriented.

5. System warnings to the user that a polygon might be too thin, contains an excessive number of vertices, or lacks required vertices (at "T"-type edge intersections).

6. Conversion of the graphics component to standardized graphics software (CORE or DI-3000).

7. Conversion of the interactive input component to standardized input (CORE or DI-3000), and possibly the Polhemus.

8. Conversion of the internal polygon database to SurfsUP procedures.

It is worthwhile noting that all of these features are more-or-less available through SurfsUP. For example, our CORE now supports direct vertex picking from the screen, IntSurf has a general *sweep* facility which could replace the BUILD *surface of revolution*, and IntSurf also has general polyhedral intersection and union routines which could possibly replace BUILD *mili* and *punch*. It appears that most of the remaining BUILD functions are available rather directly through IntSurf, though clearly without benefit of a graphical interface.

It is our opinion, therefore, that it would be easier to convert the functionality of BUILD into the SurfsUP environment than to attempt to duplicate the SurfsUP capabilities in continued *ad hoc* extensions of the current BUILD. This effort could be contracted out, but it might also be performed by available staff at JSC.

One open issue in this discussion is the impact of external CAD systems (such as MCAUTO) on the design and encoding of polyhedral structures. Our perception is that such turnkey systems are not expected to replace BUILD at NASA JSC in the near future, though an increasing number of object models may become available to OSDS in this fashion. If this is indeed the case, the issue of a new BUILD may be secondary to database conversion problems in general.

7. Anthropometric Lab Integration

Our discussion of the Anthropometric Lab (AML) integration will focus on three issues: anthropometric measurement databases, reach databases, and body segment shape utilization.

7.1. AML Measurement Databases

There are three categories of information that the AML could obtain that would be usable by TEMPUS: anthropometric (size) measurements, joint limit data, and strength data. The first two have corresponding database structures in TEMPUS which have been reported on in earlier Progress Reports. The third, strength data, is discussed at length in a separate report [2], though its integration will also be addressed here.

As presently constructed, all anthropometric parameters such as joint limits and regression formulas are embedded in program code or files. In order to integrate the AML data collection process into OSDS and TEMPUS, methods must be developed for interactively obtaining AML data and transforming it into appropriate databases and formulas. TEMPUS already has features to select among varying anthropometric databases and joint limit databases. Keeping these databases updated could be expedited by using a true database system (such as the VAX RDB system) to store individual and generic anthropometric data. New information would be entered through the database manager (or through TEMPUS, as is presently done) and the necessary population statistics would be updated. There are no significant technical problems in this process, only the *responsibility* must be placed on a JSC staff member to manually update the TEMPUS database as new data is collected.

In light of our recommendations for human strength models, the database approach is also valid for this data [2]. Since the strength data would be indexed (at least) by body part and joint angle, strength information as collected can be entered directly into the database. Again, someone must be repsonsible for data entry and consistency, but much of the essential information may be obtained from published data or (preferably) direct experiment. Since the database approach permits strength data for individuals as well as populations, the unique characteristics of AML subjects may be preserved and utilized.

7.2. Reach (Workspace) Databases

There are three components to the creation and maintainance of reach databases:

1. Creating workspaces empirically (via PLAID REACH) or analytically (via Jim Korein's workspace generation code [5]).

2. Creating, storing, retrieving, and utilizing reach data and workspaces in TEMPUS.

3. Integration of both empirical and analytic databases.

Presently the reach workspace generation code is operable but not integrated directly into TEMPUS. It is available through IntSurf and some additional testing and development is needed. The code to perform a revolute sweep (and compute the corresponding boundary) works on *psurf* objects of low to medium complexity, but more complex figures must be tested. Some preliminary testing has also been done on the spherical sweep of *psurf* objects. The testing of complex spherically swept objects will follow when the revolute sweep is proved to be reliable.

One significant problem which has not yet been addressed is the overall efficiency of the sweep and union operations. Presently, the computation cost (time) involved in generating workspaces is quite high. It would be very advantageous to make this process more efficient. This is particularly true with complex workspace generation (*i.e.* chains of workspaces). There are a number of possibilities in this regard, none of which are trivial. One might be to make sure that the resulting workspace from each sweep is simplified as much as possible (perhaps at the user's discretion) before the workspace is used as the next *psurf* object to be swept. Another possibility might be to increase the efficiency of the face intersection routine (where the resulting *psurf* is "self-intersected"). Face intersection consumes a considerable part of the effort needed to generate complex workspaces.

One direction which we have not pursued though it is worth mention is the representation of reach spaces as volumes rather than surfaces. The advantage to this approach is that the union operation becomes nearly trivial; the disadvantage is the (typically) great storage requirements if considerable spatial detail (resolution) is required. Methods used include various spatial subdivisions such as oct-trees or voxels. Present technology limits these representations to relatively low resolution data effectively 256 points in each dimension, for example [8].

If workspaces are indeed stored as polyhedra (or even as volumes), the databases for such information will be extremely large. Data for particular individuals might need to be stored in separate, possibly even off-line, files. This database must be pre-computed during slack usage periods because of the extreme overhead involved. Otherwise there are no major problems involved in storing, retrieving, using, or displaying reach workspace data.

Given either a polyhedral or volume representation of an analytically-derived reach workspace, the geometric region may be compared graphically with data derived empirically via PLAID's *Contour* function. If there is close agreement, wonderful. More likely, however, the empirical and analytic data may disagree. In this case the stored information can be biased towards a liberal estimate by taking the union of the two spaces, or toward a conservative estimate by taking the intersection. Both operations are supported in SurfsUp.

7.3. Body Segment Databases

There are several topics that should be addressed to improve the shape and structure of the bodies.

1. Acquisition of body surface data on a segment by segment basis.

2. Body surfaces which depend on joint parameters or body states, *e.g.* upper arm under elbow flexion/extension; torso under breathing. This involves interpolating between key shape positions. Different somatotypes can also be constructed by this method.

3. Multiple levels of detail: BUBBLEpeople, Polybody, "Mr. T," and stickman; selected by user choice or by image size on display.

We examine each of these in the following sections.

7.3.1. Segment Shape

TEMPUS capabilities in the area of segment shape have evolved slowly and we realize that more effort is required. In particular, the body segments are created manually; there are no automatic methods for computing segment shape from actual surface data points. We expect that an existing convex hull algorithm will be useful, given surface points on a convex segment. Fortunately, most body segments are roughly convex. The surface points that lie inside a concavity may have to be "manually" connected into the surface net in order to accurately portray the shape. One issue with any approach to this problem is the use of such an accurate segment boundary. If its use

is for graphical display, then any reasonable approximation will do. If its use is for collision detection, then an accurate surface is more important. Note, however, that the general flexibility and resiliency of body surfaces (due to the internal skeleton and fleshy muscle masses) may render accurate surface models unduly restrictive. People are able to tolerate a certain level of discomfort while pressing their segments beyond their nominal shape. Thus the entire problem would seem of low priority.

7.3.2. Spine Bending and Torso Breathing

The work done so far in the area of segment shape adjustment according to local body configuration is somewhat more important. In particular, our efforts have been directed toward modifying the spine in the TEMPUS body to permit smooth bending and to scaling the torso spheres to simulate the effects of breathing. In particular, the BUBBLEPEOPLE display is varied in such a way as to simulate torso bending and breathing. The primary modification made was the variation of the size and shape of the torso segments based upon the depth of the breathing and the position in the breathing cycle. Also we incorporated a (temporary) method to simulate the raising and lowering of the shoulders which typically occurs with breathing.

A module (*DRBUBFIG*) which had previously existed only as a skeleton was fleshed out to display BUBBLEPEOPLE by traversing all of the segments of the body and performing several manipulations on each of the spheres in every segment which had to be redrawn. Since each segment has its own coordinate system, every sphere to be drawn must first be converted to the world coordinate system. When this is done, the sphere must be converted to a polygon (currently 12-sided) which approximates the silhouette the sphere would produce from the current viewing direction.

Once BUBBLEPEOPLE could be displayed in TEMPUS we were able to convert the old BODTEST capability of modeling a flexible spine. Since the TEMPUS body spine is only represented by its two ends, everything between these points was assumed to be straight. When a body is standing straight, this is fine. When the waist is bent, however, the resulting figure looks very rigid and can develop discontinuities along its sides. The curvable spine routines model the spine as a spline curve interpolated between the waist and neck. It is important to note that this feature does not entail a change in the representation of the body and only modifies the display of the spheres.

The routines which performed the spine curve calculations had to be modified

slightly to conform to TEMPUS conventions. The routines formerly just calculated the spine curve and drew the spheres in the torso segments offset from this curve depending on their relative position to a straight spine. Since *DRBUBFIG* is already being used to draw the spheres, the routines in *BODYSPINE* had to be changed to take a sphere as a parameter and return a new sphere offset from the spine curve without displaying anything. *DRBUBFIG* then only had to call the routine to first calculate the spine curve and then pass the spheres of the torso segments to the procedure (*PROJTORSOSPHERE*) which translated the spheres appropriately. The resulting display after the modifications were made was a BUBBLEWOMAN whose torso segments curved smoothly as the waist bent.

We were now able to formulate the changes in torso segment girth appropriate to breathing. First we had to acquire some rough data approximating the changes in girth of the torso as a person inhales and exhales. For the sake of generality, all the data were calculated relative to the torso size so that the numbers would represent percentage deviation from a normal (relaxed) position. Data were obtained for only three locations representing the tops of the upper, center, and lower torso.

Location	Scale Factor
Top of Upper Torso	0.036
Top of Center Torso	0.045
Top of Lower Torso	0.008

These data represent the maximum percentage change in the torso size, and it was assumed that the maximum increase during inhale was the same as the maximum decrease during exhale. The only surprising fact is that the top of the center torso changes slightly more than the top of the upper torso.

The first idea to simply use these values as the scaling factors of each torso segment was rejected because of the possibility that discontinuities could develop along the boundaries between segments. A way was needed to be able to scale the spheres continuously along the entire torso, and having incorporated the curved spine routines provided just the needed knowledge. Since the spheres returned by the curved spine routine (*PROJTORSOSPHERE*) were all in the lower torso's coordinate system, the z-value of the spheres could be used to calcuate the relative distance of the spheres from the bottom of the lower torso (RelZ = 0) to the top of the upper torso (RelZ = 1). However, it was decided that the scaling for breathing should be done before the spine curve transformations, so a function was written (*GetzRelTorso*) which returned the relative z-value given the actual z-value and the segment in which the sphere resides.

Using this idea, a function was written which returns an appropriate scaling factor depending on the relative z-value of the sphere. The method used was a simple linear interpolation of the scale factors between the known values. The value for the bottom of the lower torso was assumed to be zero.

To increase the efficiency of this system, a procedure (*BreathInit*) was written which would only have to be called once for each display of a body. This procedure takes two parameters, the depth of the breath (from 0=none to 1=maximum) and the position in the breath cycle (an angle in degrees), and from these it calculates many variables which will be used repeatedly for all of the spheres in the torso segments. Using *BreathInit* to calculate frequently used expressions only once for each display ensures that at most one floating point multiplication and two floating point additions will be needed for each sphere to scale the torso segments.

To model the raising of the shoulders which normally accompanies breathing, *BreathInit* also calculates the amount by which the upper torso is raised or lowered. Another procedure (*BrArmRaise*) is then passed all spheres in the segments representing the arms and the hands. This procedure simply changes the height (z-value) of these spheres by the amount last calculated in *BreathInit*. This method, however, is temporary since it has some definite shortcomings. Since the entire arm is raised (or lowered) slightly, the hand will raise and lower as well. This will normally cause no problems, but if the hand were meant to be stationary on some object in the scene, one might become annoyed to see that in the animated version of the scene, the hand is drifting up and down past the object. One way which this problem could be fixed would be to raise and lower the arms by changing the joint angles of the clavicles. If this were done, then when a person wants to be sure that the hand stays in one position, both key frames would have to place the hand at the proper place using a position reach. Then, although the shoulders may be going up and down between these key frames, the TEMPUS animator would interpolate the position reaches with other position reaches (which depend on the shoulder position), thereby keeping the hand where it belongs.

The maximum scale factors for breathing have been left slightly higher than the data values obtained because the user can always select a depth of breathing less than the maximum, but if the maximum were set too low, the user could not select a depth higher that that value without recompiling the breathing module and relinking TEMPUS.

Since it had been decided that the breathing changes should only affect the display and not change the actual stored representation, the breathing parameters currently only affect the display of BUBBLEPEOPLE. This is not much of a shortcoming because it is unlikely that the other body representations (STICK figures and POLYBODY figures) would benefit from the added realism of breathing. There is no way to get the torso segment of a STICK figure to expand (it is only a line), and POLYBODY figures already are a coarse surface approximation.

7.3.3. Multiple Levels of Detail

Our work on multiple levels of detail in TEMPUS has been very valuable. There should be a mechanism in TEMPUS for automatically selecting the most efficient model based on its projected image size on the display screen. The techniques to do this are simply based on determining image height sizes where one representation is to be changed to another. TEMPUS already supports manual changes in body display type, so the automatic feature would not be difficult to add.

8. Support and Maintenance

We have been providing ongoing support and maintenance for all software provided to NASA under the terms of contract NAS9-16634. In the long run it is in NASA's best interests to transfer some of the TEMPUS code to the jurisdiction of local JSC support staff. The most likely components are the graphics interface (through DI-3000, if possible), and the geometry system SurfsUP. We see the latter as a possible base for future PLAID and especially BUILD systems. There is still work that can be done on SurfsUP to make it a production system, most notably the incorporation of better (*i.e.* relative rather than absolute) error-bound checking, but it is basically a stable, debugged, and highly capable hierarchic object representation system. We believe that SurfsUP can more than adequately substitute for most of the geometric code extant in PLAID. While its existance in PASCAL code might be deemed a disadvantage, it seems hardly worth the effort to re-write its highly structured environment as flat FORTRAN arrays. Very significant resources were dedicated to SurfsUP, and we would hope that the results are usable outside the TEMPUS system.

There are numerous projects that we would like to do in order to turn TEMPUS into the highly flexible and useful system we have envisioned it to be. These include both the introduction of new features as well as the repair or maintenance of existing ones. Among the priority items on our list are:

- Improvements in reach and positioning:
  - Implement *affixment* commands. These will allow degrees-of-freedom of one object to be attached, at the user's discretion, to other objects. Thus the motions of objects can be constrained to one another, such as when a movable object is grasped in the hand.
  - Position/orientation reach. The orientation component has not been adequately integrated into TEMPUS yet.
  - Positioning aids using object faces, vertices, and edges (*e.g., line up two faces, place on, etc.*). These can be used for many purposes, including *sit* and *stand on* type commands.
  - Put in actual limits for joint potentiometers.
  - Incorporate the six Polhemus digitizer input values into the reach parameter specification.
  - Allow optional picking of moved object type and reference frame as part of each global motion command.
- Macros
  - Allow the user many options in the way macros are run, including a *step* option, *abort on error* option, *bell off* option, *etc.*
  - Make macros and picture files impervious to obsolescence.
  - Make macros editable.
- Menus
  - Provide the user with an hierarchical list for fast workstation part picking.
  - Generalize menu systems (top level and device) to be more truly device independent.
  - Allow sets of items to be selected from list menus. This is especially important for the extended clearance command, which needs a set of body segments as input.
- Shaded
  - Adapt clipping to SolidView output.
  - Adapt object decomposition routines for routines that need them (Lexidata SolidView output, *shaded* output, clearance routines, *etc.*).
- View
  - Put in a better *person's view* command. This involves more accurate determination of the parameters of people's views and ways of showing

these (*e.g.*, foveal fields of view, setting of degree of perspective, *etc.*) to the user.

- Put in a view saving and restoring command.

- Workspace generation and use.

- Assorted bug extermination.

9. Impact of UNIX and Local Workstations

We have been investigating the operating system environment for TEMPUS on a VAX to assess the impact of UNIX and UNIX-based workstations. The present facts in this case are noted:

- TEMPUS presently consists of over 5000 Pascal modules. PLAID is a system of commensurate scope written in FORTRAN. Neither FORTRAN nor Pascal has a satisfactory compiler in the VAX environment. In the case of FORTRAN, the resulting code is significantly slower than for VMS FORTRAN. For Pascal, the essential software engineering tools we have come to rely on are not present (such as Modules and Environments) and the Pascal is only ISO standard (a restriction for us). The effort required to convert to UNIX Pascal would be considerable and essentially non-productive.

- There are no plans that we know of to have VMS Pascal converted to run under the VAX UNIX environment. This may occcur, either through actions of DEC or a third party software vendor, but we are unaware (through the Vax Users' Group) of any current effort.

- The conversion of TEMPUS to C would be a major two year undertaking which would serve no useful purpose to anyone and only retard progress and research in essential areas for all of us including the contract officers.

- Distributing some of the functions of TEMPUS (and PLAID) to UNIX workstations is feasible, desirable, and even underway. For example, the graphics visible surface rendering system we have obtained from Frank Crow is written in C and runs on a UNIX system. This code is running right now on the UNIX VAX in the GRASP Lab here in our Department. Also, the real-time animation software is being written for the IMI 500: a UNIX workstation which drives a high performance vector display system.

- Given that high-speed communication between computers running different operating systems (*e.g.* VMS and UNIX) is not difficult using existing networks (*e.g.* Ethernet) and protocols (*e.g.* TCP/IP), there is no reason to oppose migration of various software components onto a variety of network-compatible machines. They can communicate as needed by shared files or actual file transfer. The major benefits of this arrangement are the possibilities of using each machine to advantage with particular software, and the overall advantage gained from parallel (distributed) processing. Thus as long as there is at least one network node running VAX VMS, the other

computers in the network may be utilized for graphics rendering, real-time motion display, database storage and management, user input processing, *etc.*

On the basis of these facts we recommend the following:

1. At least one of the Crew Station Design Section VAX computers remain running VMS for the forseeable future,

2. Any conversion to UNIX be made in local workstations running particular subprocesses or tasks of the overall OSDS software such as shaded graphics or real-time animations.

3. UNIX workstations may be used profitably and effectively for system software development as they provide a flexible and usually graphically-oriented interface to the programmer.

4. All computers should be tied together on a local area network, preferably Ethernet, in order to accomodate the volume of information flow between machines necessitated by the transfer of graphical data and physical object databases.

5. Choice of workstations should reflect the types of activities to be factored out of OSDS: for example, VAXSTATION II to run existing VAX software, IMI 500 or Silicon Graphics IRIS 2400 to run real-time line drawing animation graphics, Sun or Apollo workstations for general software developement, *etc.*

10. Conclusions

The development of TEMPUS and its integration with PLAID into a powerful and flexible OSDS facility will require dedicated resources over the next few years. The tasks ahead, however, appear to be mostly clear and the evolutionary directions consistent with concurrent trends in computer and computer graphics technology. In particular, there are three general thrusts in the system integration area:

- Better interfaces between TEMPUS and PLAID both for geometric objects and graphics input and output.

- Better databases for a wider range of anthropometric data such as segment sizes, reach spaces, and strength data.

- Better utilization of computer resources including distributed computation and graphics workstations.

The the TEMPUS and PLAID interface we must evolve a common database or at least user-transparent conversions between geometric objects created in PLAID and manipulated in TEMPUS. Both systems should rely on the same standard CORE-like graphics input and output code, and may eventually share a common user interface. The TEMPUS system should continue to evolve necessary task analysis interfaces and

interactive enhancements. The shaded graphics systems should be based on externally-supported systems if at all possible, since such products are apt to become readily available within the next few years as competitive systems emerge. The geometric subsystem of TEMPUS, SurfsUP, could be adapted for use in later versions of PLAID. A more natural interactive object building and design system would be desirable, including high-level design aids to check object validity during the data generation process.

In the anthropometric area, there should be movement toward use of a supported database system such as DEC's RDB for all AML and strength data. Such a system would have its own database manager and would facilitate the regular updating of TEMPUS anthropometric data. There must be someone identified at JSC as being responsible for data entry and validity. Reach data should be analytically computed for common body chains and geometrically compared and combined with empirically-obtained data from SELSPOT data and PLAID's *Contour* program. Body modeling from the database should be improved to handle the automatic selection of display method from the multiple levels of detail available in TEMPUS. The bodies themselves should be enhanced with full girth and somatotype scaling along the lines of the efforts done for torso bending and breathing.

Finally, the massive sizes of TEMPUS and PLAID motivates the move to distributed systems to take advantage of parallel computations and special purpose programs. For example, the TEMPUS animation system and the shaded graphics rendering systems can run independently from the primary TEMPUS program, yet communicate with it to share files and pass commands and parameters. VAXSTATION II, Ethernet communications, and high-performance, real-time graphics workstations such as the IMI 500 and the Silicon Graphics IRIS 2400 are prime examples of the type of available systems that will mold the shape of the OSDS of the late 1980's.

11. Schedule and Resources

The tasks outlined in the Conclusions could be realized over a three year period if suitable personnel were directed to its implementation. The schedule would, of course, differ if other directions were taken. In particular, tighter integration will take longer initially but will pay off eventually in easier, coordinated development. Also, there should be a long term plan to provide ongoing maintenance for TEMPUS during its lifetime. The approximate timetable for the systems integration effort is given in Table

11-1.

Table 11-1: Systems Integration Schedule

Time Milestone	Task (per staff member)
year 0.5	TEMPUS to PLAID object interface and vice versa. DI-3000 and Shaded graphics interfaces. Convert to database system (RDB) for AML and strength data.
year 1	Improve TEMPUS interactive interface. Interactive BUILD using Polhemus. Full girth and somatotype scaling; multiple levels of detail.
year 2	Complete animation system. Complete TEMPUS reach databases. TEMPUS running on VAXSTATION II in distributed system.
year 3	Geometric object input validity testing. Integrate TEMPUS and PLAID reach databases. Common input and graphics system for PLAID and TEMPUS

The *time milestone* is the length of time from project inception (not a duration) to the completion of the indicated *tasks*. The tasks are a summary of the work needed to fulfill the system requirements discussed in the Conclusion. Each task refers to either the *full-time* Research Specialist or *one* graduate research assistant. For the latter, this is a half time load (20 hours/week). Thus multiple tasks for one time milestone are assumed to proceed in parallel, and a total of three individuals for three years are required.

The resources required are summarized in Table 11-2. The monetary estimates are based on solely on 1985 University of Pennsylvania rates including employee benefits, tuition, and overhead as applicable. There is no provision for inflation; that may be projected by NASA as necessary.

Table 11-2: System Integration Resources

1 Research Specialist	$61K/year
2 Graduate Research Assistants for duration of project	50K/year
Faculty supervision time (10% of academic year)	10K/year
Equipment:	
VaxStation II	20K
Travel, current expense, duplicating, etc	34K/year
Totals:	
Year 1:	$175K
Year 2:	$155K
Year 3:	$155K

12. Bibliography

1. Norman I. Badler, Paul Fishwick, Nina Taft, and Mukul Agrawala. Zero-Gravity Movement Studies. Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, 1985. (For NAS9-16634).

2. Norman I. Badler, Philip Lee, and Sui Wong. Strength Modeling Report. Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, 1985. (For NAS9-16634).

3. Lynne S. Brotman and Norman I. Badler. "Generating Soft Shadows With a Depth Buffer Algorithm". *IEEE Computer Graphics and Applications 4*, 10 (October 1984), 5-12.

4. Frank C. Crow. "A more flexible image generation environment". *Computer Graphics 16*, 3 (July 1982), 9-18.

5. James U. Korein. *A geometric investigation of reach*. MIT Press, Cambridge, MA, 1985.

6. M. H. Liewald and P. R. Kennicott. "Intersystem data transfer via IGES". *IEEE Computer Graphics and Applications 2*, 3 (May 1982), 55-63.

7. National Bureau of Standards. Experimental Solids Proposal.

8. R. A. Reynolds. *Fast methods for 3-D display of medical objects*. Ph.D. Th., Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, 1985.

9. P. R. Wilson, I. D. Faux, M. C. Ostrowski, and K. G. Pasquill. "Interfaces for data transfer between solid modeling systems". *IEEE Computer Graphics and Applications 5*, 1 (January 1985), 41-51.