**SANDIA REPORT**

SAND2009-3170
Unlimited Release
Printed May 2009

# IceT Users' Guide and Reference

Kenneth Moreland

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.



Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.



Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2009-3170
Unlimited Release
Printed May 2009

# IceT Users' Guide and Reference

Kenneth Moreland
Data Analysis and Visualization
Sandia National Laboratories
P.O. Box 5800 MS 1323
Albuquerque, NM 87185-1323
kmorel@sandia.gov

**Abstract**

The Image Composition Engine for Tiles (IceT) is a high-performance sort-last parallel rendering library. In addition to providing accelerated rendering for a standard display, IceT provides the unique ability to generate images for tiled displays. The overall resolution of the display may be several times larger than any viewport that may be rendered by a single machine. This document is an overview of the user interface to IceT.

# Acknowledgement

I would like to thank Brian Wylie. It was his “big ideas” that got the ball rolling on the IceT algorithms and library, and it was his continuing vision that pushed us on this path to parallel rendering.

I would also like to thank the folks at Kitware, Inc. for adopting the IceT library as the parallel rendering library for ParaView. They also maintain the IceT code repository. Without them, IceT would probably be collecting dust on a crashed RAID somewhere.

# Contents

# List of Figures

# Chapter 1

# Introduction

The Image Composition Engine for Tiles (IceT) is an API designed to enable OpenGL applications to perform Sort-Last parallel rendering on very large displays. The displays are assumed to be tile displays. The overall resolution of the display may be several times larger than any viewport that may be rendered by a single machine. It is also assumed that several processes in the parallel application are **display processes**. That is, their entire display window makes up part of the display.

The design philosophy behind IceT is to allow very large sets of polygons to be displayed on very high resolution displays. As such, fast frame rates are sacrificed in lieu of very scalable and very high polygon/second rendering rates. That said, there are many features in IceT that allow an application to achieve interactive rates. These include image inflation, floating viewports, active pixel encoding, and data replication. Together, these features make IceT a versatile parallel rendering application that provides near optimal parallel rendering under most data size and image size combinations. As an example, the ParaView application[1] is using IceT for all of its parallel rendering needs ranging from a desktop sized image to the world's largest tile displays and from polygon counts ranging from 1 to 1 million (and growing).

IceT is designed to take advantage of **spatial decomposition** of the geometry being rendered. That is, it works best if all the geometry on each process is located in as small a region of space as possible. When this is true, each process usually projects geometry on only a small section of the screen. This results in less work for the compositing engine. This is of particular importance for displays with a large number of pixels.

IceT can also be used to perform sort-last parallel rendering to a single display. Such **single-tile rendering** is simply a special case of the multi-tile display IceT was designed for. Many of the optimizations done by IceT apply to the single-tile mode. Using IceT for this purpose is quite worthwhile. IceT's performance should rival that of other such software image compositors.

The rest of this document describes the use of the IceT API. There are also separate manual pages for each of the functions described here. For more details on IceT's algorithms, see:

> Kenneth Moreland, Brian Wylie, and Constantine Pavlakos. "Sort-last parallel rendering for viewing extremely large data sets on tile displays," In *Proceedings of IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, October 2001,

[1] http://www.paraview.org

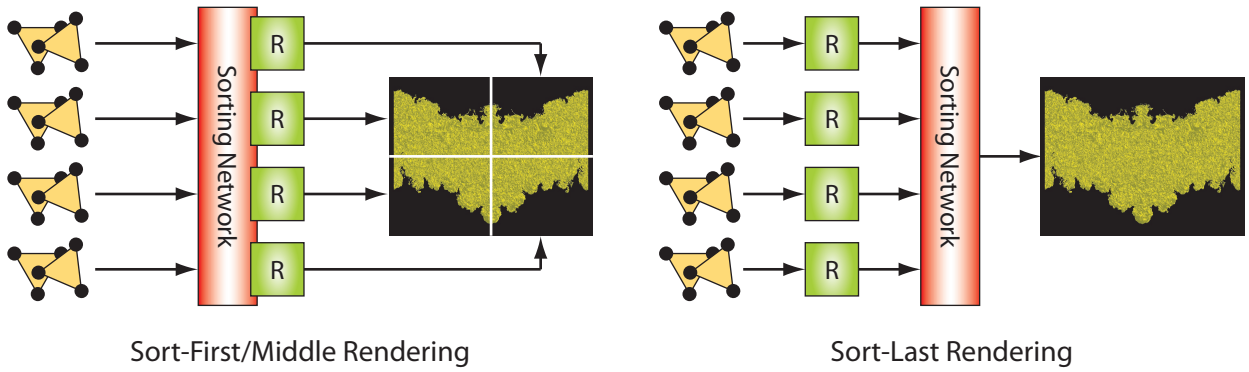


**Figure 1.1.** The differences between parallel rendering classes. Sort-first and sort-middle algorithms transfer geometric data. Sort-last algorithms transfer image data.

pp. 85–154.

# A Parallel Rendering Primer

IceT requires you to know very little about parallel rendering and their algorithms. However, it is helpful to know the basic idea behind IceT's algorithms. This section gives a brief introduction to how IceT renders in parallel.

Parallel rendering algorithms are classified as **sort-first**, **sort-middle**, or **sort-last**. The key distinguishing feature of each class is how primitives are distributed amongst processes. As demonstrated in Figure 1.1, sort-first and sort-middle algorithms allocate screen space to processes and send the appropriate geometry to each process every frame whereas sort-last algorithms render static partitions of geometry in each process and then composite the resulting images to a single image.[2] IceT is a sort-last parallel rendering library.

A convenient feature of sort-last rendering is that an application needs to change very little about how it renders geometry. The geometry is rendered the same in parallel as it is in serial; the only difference is that each process only renders a subset of the geometry. The typical operation of a parallel application using sort-last rendering is to simply render locally and then composite the images.

When rendering to a tiled display, as IceT allows you to do, there is an added level of complexity introduced because the graphics system is often not capable of rendering an image large enough for the entire display. Thus, image compositing for a tiled display requires a loop that can

[2]In the interest of brevity and clarity, I am intentionally leaving out details that are unimportant to understanding IceT such as hybrid algorithms and differences between sort-first and sort-middle algorithms.

iteratively render images for each tile and composite them. IceT handles this looping and interfaces with the rendering functions of your application through a callback mechanism. This will be described in the following chapters.

# Chapter 2

# Tutorial

In this chapter we outline the steps required to create a simple IceT application from building the IceT source, using the created libraries, and writing your own applications. IceT is solely responsible for the image composition part of parallel rendering. Thus, it relies on two other APIs: **OpenGL** for rendering and a communication layer for passing messages such as **MPI**, the Message Passing Interface. Both have implementations in nearly every computer architecture.

This tutorial assumes the reader is familiar with OpenGL. If this is your first experience with OpenGL programing, consider trying some typical serial rendering before jumping into the parallel rendering domain. A familiarity with MPI is also helpful.

## Building IceT

The IceT build process is very portable. It can be compiled on Microsoft Windows, Macintosh OS X, and a wide variety of Unix implementations. IceT can be built on just about any platform that has an OpenGL 1.1 compliant installation. Most modern operating systems come distributed with OpenGL. For those that are not, you can usually use the **Mesa 3D** library (www.mesa3d.org), a software implementation of OpenGL. An installation of MPI is also almost always needed, although not strictly required. **MPICH** (http://www-unix.mcs.anl.gov/mpi/mpich2/) is a free and widely portable implementation of MPI.

IceT uses **CMake** to build across so many different platforms. As such, you will have to download the CMake build tools from www.cmake.org and install. Then, create a build directory and run the CMake program (from the "Start" menu on Windows or ccmake on Unix and Mac OS X). CMake will determine the parameters of your system and do its best to find libraries on which IceT depends. The CMake program, shown in Figure 2.1 will also provide a GUI to allow you to easily change build parameters and external libraries.

CMake will generate a set of build files for the local system. The type of files depends on the type of machine you are using and the compile system you have chosen to use. On Unix machines, make files are the most common. On Windows, you usually generate MSVC project files or nmake files. On Mac OS X, either make files or Xcode project files are commonly generated based on user selection. You then use the native build system to build and, optionally, install IceT.

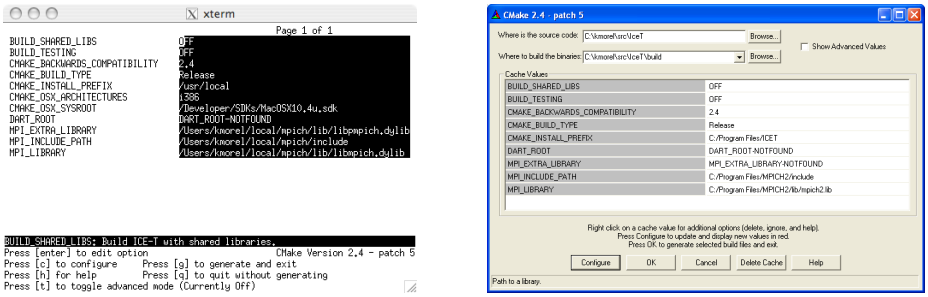


**Figure 2.1.** The CMake user interface. The Unix version is on the left whereas the Microsoft Windows version is on the right.

# Linking to IceT Libraries

IceT comes with three libraries: **icet**, **icet_strategies**, and **icet_mpi**. The actual filenames of these libraries varies depending on the filesystem and build type. For example, on most Unix systems, a static build results in filenames of libicet.a and the like whereas shared libraries are libicet.so. Windows has libraries with names like icet.lib as well as icet.dll if building shared libraries. However, the difference in these filenames usually hidden by the build system, especially if you use a portable build system like CMake.

You are, of course, free to use whatever build system you like, whether it be system specific or cross platform. Using IceT is simply a matter of finding the header and library files. However, because IceT is built with CMake, it comes with some extra facilities for helping other CMake builds find it. This section will give you the bare minimum you need to set up CMake to build an application using IceT. Readers interested more about CMake should pick up a copy of *Mastering CMake* by Ken Martin and Bill Hoffman.

You define a build system with CMake by creating a **CMakeLists.txt** file. The CMakeLists.txt file is basically a simple script that gives commands the CMake to tell it how to build your project. Most CMakeLists.txt files start with the `PROJECT` command, which associates a name with your project and optionally specifies a language.

```
PROJECT(IceT_Tutorial)
```

Distributed with the IceT source code is a file called **FindIceT.cmake** that provides all the CMake facilities needed to find and use an IceT build. It works by finding a file called **ICETConfig.cmake**, which is written when IceT is built and contains all the necessary build settings. The FindIceT.cmake script can be invoked with the `FIND_PACKAGE` command. After the IceT package is found, a variable named `ICET_USE_FILE` is set to a file that may be `INCLUDE`d in your project to point it to the directories containing the header and library files.

```
FIND_PACKAGE(IceT REQUIRED)
```

```
INCLUDE(${ICET_USE_FILE})
```

Any application using IceT will also be using OpenGL and almost all will be using MPI. In addition, the example in the following section also uses GLUT for window management. CMake comes with modules to find all three of these libraries, which makes it easy to include in our project.

```
FIND_PACKAGE(OpenGL REQUIRED)
FIND_PACKAGE(GLUT REQUIRED)
FIND_PACKAGE(MPI REQUIRED)

MARK_AS_ADVANCED(CLEAR
  MPI_INCLUDE_PATH
  MPI_LIBRARY
  MPI_EXTRA_LIBRARY
  )

INCLUDE_DIRECTORIES(
  ${OPENGL_INCLUDE_DIR}
  ${MPI_INCLUDE_PATH}
  ${GLUT_INCLUDE_DIR}
  )
```

The only think left to do is to tell CMake to build a program from a set of sources and libraries specified with the `ADD_EXECUTABLE` and `TARGET_LINK_LIBRARIES` commands, respectively.

```
ADD_EXECUTABLE(Tutorial Tutorial.c)
TARGET_LINK_LIBRARIES(Tutorial
  ${OPENGL_LIBRARIES}
  ${GLUT_LIBRARIES}
  ${MPI_LIBRARY}
  ${MPI_EXTRA_LIBRARY}
  ${ICET_CORE_LIBS}
  ${ICET_MPI_LIBS}
  )
```

# Creating IceT Enabled Applications

To use IceT, include it's header: `GL/ice-t.h`. You will almost always need to also include the header containing an MPI version of an IceT communicator: `GL/ice-t_mpi.h`. On the rare occasion that you need to use IceT with a communication layer other than MPI, you can define a custom communicator as described in Chapter 6.

```
#include <GL/ice-t.h>
#include <GL/ice-t_mpi.h>
```

Before you call any IceT functions, you need to initialize MPI by calling `MPI_Init`. You will also need to create an OpenGL context (that is, open an OpenGL window). Do this by first creating an IceT **communicator** from an MPI communicator and then using that to create an **IceT context**.

```
comm = icetCreateMPICommunicator(MPI_COMM_WORLD);
context = icetCreateContext(comm);
```

In the proceeding code, `comm` is of type `IceTCommunicator` and `context` is of type `IceTContext`.

Now that we have created and activated an IceT communicator, as well as initialized the IceT **state**, we can start using IceT. It is often useful to first query IceT on the size of the parallel job it is running in and what is the local process id, or **rank**. The values are stored in variables of type `GLint`.

```
icetGetIntegerv(ICET_RANK, &rank);
icetGetIntegerv(ICET_NUM_PROCESSES, &num_proc);
```

In addition to an IceT context, you will also need an **OpenGL context**. In other words, you need to make the rendering window in which the OpenGL rendering commands will go. The process for doing this is greatly dependent on the windowing system and beyond the scope of this document. It is usually easiest to use a third party API to do this. If you are not already using a GUI tool that generates OpenGL windows for you, then the **GLUT** API is a popular choice for simple applications.

Before rendering, we need to tell IceT the layout of the tile display using the **`icetResetTiles`** and **`icetAddTile`** functions. These commands must be executed with the same arguments on all processes of the parallel job. IceT will assume that you setup the same display layout everywhere.

If you are not actually driving a tile display and instead just generating a desktop-sized image, the following commands will correctly establish the IceT state.

```
icetResetTiles();
icetAddTile(0, 0, WINDOW_WIDTH, WINDOW_HEIGHT, 0);
```

The **`icetResetTiles`** function simply tells IceT that you are about to define a display layout. Each call to **`icetAddTile`** defines a tile in the display. In the case of a single image, the **single-tile rendering** mode, **`icetAddTile`** is called only once. The first two arguments to **`icetAddTile`** have no effect in this mode. The third and fourth arguments are the width and

height of the image to create. Usually you set this to the width and the height of the display window, but the Image Inflation section in Chapter 4 describes other usage for these parameters. The final argument is the rank of the **display process**. After a rendering the final complete image will available only on this process. In the example above, we have direct the image to go to process zero, often referred to as the **root process**.

To define an actual tile display, simply call the **`icetAddTile`** function multiple times. When describing tiles in a display, the first two arguments of **`icetAddTile`** describe where the lower left corner of the tile is located in respect to the overall display. All together, the first four arguments specify a viewport for the tile in an a single, cohesive high resolution display (which is what we are trying to achieve with our tile display). The code below defines a $2 \times 2$ tile display with the top two tiles displayed by processes 0 and 1 and the bottom two tiles displayed by processes 2 and 3.

```
icetResetTiles();
icetAddTile(0,           WINDOW_HEIGHT, WINDOW_WIDTH, WINDOW_HEIGHT, 0);
icetAddTile(WINDOW_WIDTH,WINDOW_HEIGHT, WINDOW_WIDTH, WINDOW_HEIGHT, 1);
icetAddTile(0,           0,             WINDOW_WIDTH, WINDOW_HEIGHT, 2);
icetAddTile(WINDOW_WIDTH,0,             WINDOW_WIDTH, WINDOW_HEIGHT, 3);
```

IceT contains several **strategies** for image composition. Changing the strategy modifies the algorithm IceT uses for parallel image compositing. You need to tell IceT which strategy to use with the **`icetStrategy`** function. The code below sets IceT to use the **reduce strategy**, which has proven to be an all-around good performer.

```
icetStrategy(ICET_STRATEGY_REDUCE);
```

Like with the display set up, all processes must set the same strategy.

IceT is almost ready to go. We just need to tell it some minimal information about how to render your geometry. First, IceT needs to know the spatial extent of the geometry to be drawn (in object space). The most natural way to do this is to use the **`icetBoundingBox`** function, which defines an axis-aligned box defined by the minimum and maximum coordinates in each dimension.

```
icetBoundingBoxf(x_min, x_max, y_min, y_max, z_min, z_max);
```

The parameters can, and should be, different on each process, since each process will have a different partition of data. Strictly speaking, identifying the geometry bounds is not necessary. If they are not defined, IceT will assume the geometry covers the entire screen. When rendering a single small image, the information is of little consequence. However, when rendering larger images this information can dramatically improve the performance of image composting. Specifying the bounds can be critical on large tile displays.

The second and final piece of information IceT needs is a way to draw your geometry. IceT achieves this through a **drawing callback**.

```
icetDrawFunc(drawScene);
```

The drawing callback is a pointer to any function that issues OpenGL commands that render geometry to the active frame buffer. The callback is free to issue most OpenGL so long as it restores all the OpenGL state (except, of course, frame buffer contents). Also, the callback function should modify neither the projection matrix nor the clear color. Care needs to be taken if the callback modifies the model view matrix. More details are given in the Drawing Callback section of Chapter 3.

IceT is now ready to render. Rendering is initiated with a call to **icetDrawFrame**. The **icetDrawFrame** must be called on all processes. The function will render the scene using the provided drawing callback, composite the image, and place the appropriate images in the back OpenGL buffers of the appropriate display processes.

```
icetDrawFrame();
```

Parallel rendering is now enabled in your application. Simply call **icetDrawFrame** every time you wish to draw a new image. The geometry rendered by your may change from frame to frame so long as you ensure that you also update IceT with the bounds of your geometry if it changes.

The following code is a full example of a simple IceT application. Do not be alarmed by the length. The majority of the code is spent in setting up the supporting libraries (OpenGL, GLUT, and MPI) and in comments.

```
/* -*- c -*- ************************************************************
** $Id: Tutorial.c 637 2008-02-18 19:35:10Z kmorel $
**
** Copyright (C) 2007 Sandia Corporation
** Under the terms of Contract DE-AC04-94AL85000, there is a non-exclusive
** license for use of this work by or on behalf of the U.S. Government.
** Redistribution and use in source and binary forms, with or without
** modification, are permitted provided that this Notice and any statement
** of authorship are reproduced on all copies.
**
** This is a simple example of using the IceT library.  It demonstrates the
** techniques described in the Tutorial chapter of the IceT User's Guide.
************************************************************************/

#include <stdlib.h>

/* IceT does not come with the facilities to create windows/OpenGL contexts.
 * we will use glut for that. */
#ifndef __APPLE__
#include <GL/glut.h>
```

```
#include <GL/gl.h>
#else
#include <GLUT/glut.h>
#include <OpenGL/gl.h>
#endif

#include <GL/ice-t.h>
#include <GL/ice-t_mpi.h>

#define NUM_TILES_X 2
#define NUM_TILES_Y 2
#define TILE_WIDTH 300
#define TILE_HEIGHT 300

static void InitIceT();
static void DoFrame();
static void Draw();

static int winId;
static IceTContext icetContext;

int main(int argc, char **argv)
{
  int rank, numProc;
  IceTCommunicator icetComm;

  /* Setup MPI. */
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numProc);

  /* Setup a window and OpenGL context.  Normally you would just place all the
   * windows at 0, 0 (and probably full screen in tile display mode) to a local
   * display, but since this is an example we are assuming that they are all
   * going to one screen for display. */
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
  glutInitWindowPosition((rank%NUM_TILES_X)*(TILE_WIDTH+10),
                         (rank/NUM_TILES_Y)*(TILE_HEIGHT+50));
  glutInitWindowSize(TILE_WIDTH, TILE_HEIGHT);
  winId = glutCreateWindow("IceT Example");

  /* Setup an IceT context.  Since we are only creating one, this context will
   * always be current. */
  icetComm = icetCreateMPICommunicator(MPI_COMM_WORLD);
  icetContext = icetCreateContext(icetComm);
  icetDestroyMPICommunicator(icetComm);
```

```
  glutDisplayFunc(InitIceT);
  glutIdleFunc(DoFrame);

  /* Glut will only draw in the main loop.  This will simply call our idle
   * callback which will in turn call icetDrawFrame. */
  glutMainLoop();

  return 0;
}

static void InitIceT()
{
  GLint rank, num_proc;

  /* We could get these directly from MPI, but it's just as easy to get them
   * from IceT. */
  icetGetIntegerv(ICET_RANK, &rank);
  icetGetIntegerv(ICET_NUM_PROCESSES, &num_proc);

  /* We should be able to set any color we want, but we should do it BEFORE
   * icetDrawFrame() is called, not in the callback drawing function.
   * There may also be limitations on the background color when performing
   * color blending. */
  glClearColor(0.2f, 0.5f, 0.1f, 1.0f);

  /* Give ICE-T a function that will issue the OpenGL drawing commands. */
  icetDrawFunc(Draw);

  /* Give ICE-T the bounds of the polygons that will be drawn.  Note that
   * we must take into account any transformation that happens within the
   * draw function (but ICE-T will take care of any transformation that
   * happens before icetDrawFrame). */
  icetBoundingBoxf(-0.5f+rank, 0.5f+rank, -0.5, 0.5, -0.5, 0.5);

  /* Set up the tiled display.  Normally, the display will be fixed for a
   * given installation, but since this is a demo, we give two specific
   * examples. */
  if (num_proc < 4)
    {
    /* Here is an example of a "1 tile" case.  This is functionally
     * identical to a traditional sort last algorithm. */
    icetResetTiles();
    icetAddTile(0, 0, TILE_WIDTH, TILE_HEIGHT, 0);
    }
  else
    {
```

```
    /* Here is an example of a 4x4 tile layout.  The tiles are displayed
     * with the following ranks:
     *
     *               +---+---+
     *               | 0 | 1 |
     *               +---+---+
     *               | 2 | 3 |
     *               +---+---+
     *
     * Each tile is simply defined by grabing a viewport in an infinite
     * global display screen.  The global viewport projection is
     * automatically set to the smallest region containing all tiles.
     *
     * This example also shows tiles abutted against each other.
     * Mullions and overlaps can be implemented by simply shifting tiles
     * on top of or away from each other.
     */
    icetResetTiles();
    icetAddTile(0,          TILE_HEIGHT, TILE_WIDTH, TILE_HEIGHT, 0);
    icetAddTile(TILE_WIDTH, TILE_HEIGHT, TILE_WIDTH, TILE_HEIGHT, 1);
    icetAddTile(0,          0,           TILE_WIDTH, TILE_HEIGHT, 2);
    icetAddTile(TILE_WIDTH, 0,           TILE_WIDTH, TILE_HEIGHT, 3);
    }

  /* Tell ICE-T what strategy to use.  The REDUCE strategy is an all-around
   * good performer. */
  icetStrategy(ICET_STRATEGY_REDUCE);

  /* Set up the projection matrix as you normally would. */
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-0.75, 0.75, -0.75, 0.75, -0.75, 0.75);

  /* Other normal OpenGL setup. */
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  if (rank%8 != 0)
    {
    GLfloat color[4];
    color[0] = (float)(rank%2);
    color[1] = (float)((rank/2)%2);
    color[2] = (float)((rank/4)%2);
    color[3] = 1.0;
    glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, color);
    }
}
```

```
static void DoFrame()
{
  /* In this idle callback, we do a simple animation loop and then exit. */
  static float angle = 0;

  GLint rank, num_proc;

  /* We could get these directly from MPI, but it's just as easy to get them
   * from IceT. */
  icetGetIntegerv(ICET_RANK, &rank);
  icetGetIntegerv(ICET_NUM_PROCESSES, &num_proc);

  if (angle <= 360)
    {
    /* We can set up a modelview matrix here and ICE-T will factor this
     * in determining the screen projection of the geometry.  Note that
     * there is further transformation in the draw function that ICE-T
     * cannot take into account.  That transformation is handled in the
     * application by deforming the bounds before giving them to
     * ICE-T. */
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glRotatef(angle, 0.0, 1.0, 0.0);
    glScalef(1.0f/num_proc, 1.0, 1.0);
    glTranslatef(-(num_proc-1)/2.0f, 0.0, 0.0);

    /* Instead of calling Draw() directly, call it indirectly through
     * icetDrawFrame().  ICE-T will automatically handle image compositing. */
    icetDrawFrame();

    /* For obvious reasons, ICE-T should be run in double-buffered frame
     * mode.  After calling icetDrawFrame, the application should do a
     * synchronize (a barrier is often about as good as you can do) and
     * then a swap buffers. */
    glutSwapBuffers();

    angle += 1;
    }
  else
    {
    /* We are done with the animation.  Bail out of the program here.  Clean
     * up IceT and the other libraries we used. */
    icetDestroyContext(icetContext);

    glutDestroyWindow(winId);
```

```
    MPI_Finalize();

    exit(0);
    }
}

static void Draw()
{
  GLint rank, num_proc;

  /* We could get these directly from MPI, but it's just as easy to get them
   * from IceT. */
  icetGetIntegerv(ICET_RANK, &rank);
  icetGetIntegerv(ICET_NUM_PROCESSES, &num_proc);

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  /* When changing the modelview matric in the draw function, you must be
   * wary of two things.  First, make sure the modelview matrix is restored
   * to what is was when the function is called.  Remember, the draw
   * function may be called multiple times and transformations may be
   * commuted.  Also, the bounds of the drawn geometry must be correctly
   * transformed before given to ICE-T.  ICE-T has no way of knowing about
   * transformations done here.  It is an error to change the projection
   * matrix in the draw function. */
  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();
  glTranslatef((float)rank, 0, 0);
  glutSolidSphere(0.5, 100, 100);
  glPopMatrix();
}
```

# Chapter 3

# Basic Usage

In this chapter we describe in greater detail the basic features of IceT. The tutorial given in Chapter 2 is a good place to start building your applications. You can then consult this chapter and later ones for more details on the operations as well as descriptions of further features.

Prototypes for the majority of IceT types, functions, and identifiers can be found in the `GL/ice-t.h` header file. You will also almost always need to include the header `GL/ice-t_mpi.h`. Chapter 6 provides more details on this latter header file's function.

```
#include <GL/ice-t.h>
#include <GL/ice-t_mpi.h>
```

## The State Machine

The IceT API borrows many concepts from OpenGL. One major concept taken is that of a state machine. At all times IceT maintains a current state. The state can influence the operations that IceT makes, and IceT's operations can modify the state.

IceT can manage multiple collections of state at the same time. It does this by associating each state with a **context**. At any given time, there is at most one active context. Any IceT function called works using the current active context.

Contexts are created and destroyed with **`icetCreateContext`** and **`icetDestroyContext`**, respectively.

```
IceTContext icetCreateContext( IceTCommunicator comm );

void icetDestroyContext( IceTContext context );
```

These functions work with an object of type **`IceTContext`**. **`IceTContext`** is an opaque type; you are not meant to directly access it. Instead, you pass the object to functions to do the work for you.

The **`icetCreateContext`** function requires an object of type **`IceTCommunicator`**. This is another opaque type that is described in more detail in Chapter 6. For now, just know

that you can create one from an MPI communicator using the **icetCreateMPICommunicator** function.

```
IceTCommunicator icetCreateMPICommunicator(
                                          MPI_Comm mpi_comm );

void icetDestroyMPICommunicator( IceTCommunicator comm );
```

The following code gives the common boilerplate for setting up your initial IceT context.

```
#include <GL/ice-t.h>
#include <GL/ice-t_mpi.h>

int main(int argc, char **argv)
{
  IceTCommunicator icetComm;
  IceTContext icetContext;

  /* Setup MPI. */
  MPI_Init(&argc, &argv);

  /* Setup an IceT context.  If we are only creating one, this context will
   * always be current. */
  icetComm = icetCreateMPICommunicator(MPI_COMM_WORLD);
  icetContext = icetCreateContext(icetComm);
  icetDestroyMPICommunicator(icetComm);

  /* Start your parallel rendering program here. */

  /* Cleanup IceT and MPI. */
  icetDestroyContext(icetContext);
  MPI_Finalize();

  return 0;
}
```

Any number of contexts may be created, each with its own associated state. At any given time, a single given context is **current**. All IceT operations are applied with the state attached to the current context. A handle to the current IceT context can be retrieved with the **icetGetContext** function, and he current context can be changed by using the **icetSetContext** function.

```
IceTContext icetGetContext( void );

void icetSetContext( IceTContext context );
```

Changing the context is a fast and easy way to swap states. This could be used, for example, to switch between rendering modes. One context could be used for a full resolution image, and

another could use **image inflation** (described in Chapter 4) to make faster but coarser images during interaction.

When a context is created, its state is initialized to default values. You can effectively "duplicate" a context by copying the state of one context to another using the **icetCopyState** function.

```
void icetCopyState( IceTContext dest,
                    IceTContext src );
```

The state of a context comprises a group of key/value pairs. The state can be queried by using any of the **icetGet** functions.

```
void icetGetDoublev( GLenum     pname,
                     GLdouble * params );

void icetGetFloatv( GLenum     pname,
                    GLfloat *  params );

void icetGetIntegerv( GLenum   pname,
                      GLint *  params );

void icetGetBooleanv( GLenum      pname,
                      GLboolean * params );

void icetGetPointerv( GLenum     pname,
                      GLvoid **  params );
```

The valid keys that can be used in the **icetGet** functions are listed in the **icetGet** documentation starting on page 114. There is no way to directly set these state variables. Instead, they are set either by IceT configuration functions or indirectly as part of the operation of IceT. The documentation for **icetGet** also describes which functions can be used to set each state entry (assuming the user has control of that state entry).

There is a special set of state entries that toggle IceT options. Although you can query this state with the **icetGetBooleanv** function, it is more typical to use the **icetIsEnabled** function. Also unlike the other state variables, these variables can be directly manipulated with the **icetEnable** and **icetDisable** functions.

```
GLboolean icetIsEnabled( GLenum pname );

void icetEnable  ( GLenum pname );

void icetDisable ( GLenum pname );
```

The options queried with **icetIsEnabled** and manipulated with **icetEnable** and **icetDisable** are listed in the **icetEnable** documentation starting on page 112.

# Diagnostics

The IceT library has a mechanism for reporting diagnostics. There are three levels of diagnostics. **Errors** are anomalous conditions that IceT considers a critical failure. An occurrence of an error generally means that the future IceT operations will have undefined behavior. When IceT is compiled in debug mode, a seg fault is intentionally raised when an error occurs to make it easier to attach a debugger to the point where the error occurred.

**Warnings** are detections of anomalous conditions that are not as severe as errors. When a warning occurs, the current operation may produce the incorrect results, but future operations should continue to work.

IceT also can also provide a large volume of **debug** messages. These messages simply indicate the status of IceT operations as they progress. They are generally of no use to anyone who is not trying to develop or debug IceT operations.

IceT diagnostics are controlled with the **icetDiagnostics** function.

```
void icetDiagnostics( GLbitfield mask );
```

The **icetDiagnostics** function takes a set of flags that can me or-ed together. The diagnostics for errors, warnings, and debug statements can be set by passing the **ICET_DIAG_ERRORS**, **ICET_DIAG_WARNINGS**, and **ICET_DIAG_DEBUG** flags, respectively. Turning on warnings implicitly turns on errors and turning on debug statements implicitly turns on errors and warnings (although there is no problem with redundantly specifying these flags).

IceT has the ability to report diagnostics either on all processes or only on the **root process** (the process with rank 0). This behavior is controlled by the **ICET_DIAG_ROOT_NODE** and **ICET_DIAG_ALL_NODES** flags. Many diagnostic messages occur on all nodes when they occur, so reporting only on node 0 can greatly reduce the number of messages with which to contend. However, messages can differ between processes or may not occur on all processes.

The special flags **ICET_DIAG_FULL** and **ICET_DIAG_OFF** turn all possible diagnostics on and all diagnostics completely off, respectively.

By default, IceT displays errors and warnings on all nodes. You can get the current diagnostic level by calling **icetGet** with **ICET_DIAGNOSTIC_LEVEL**.

# Display Definition

IceT assumes that the tiled display it is driving has each tile connected to the graphics output of one of the processes in the parallel job in which it is running. This type of arrangement is natural for any tiled display driven by a graphics cluster, and is the delivery method of many graphics APIs.

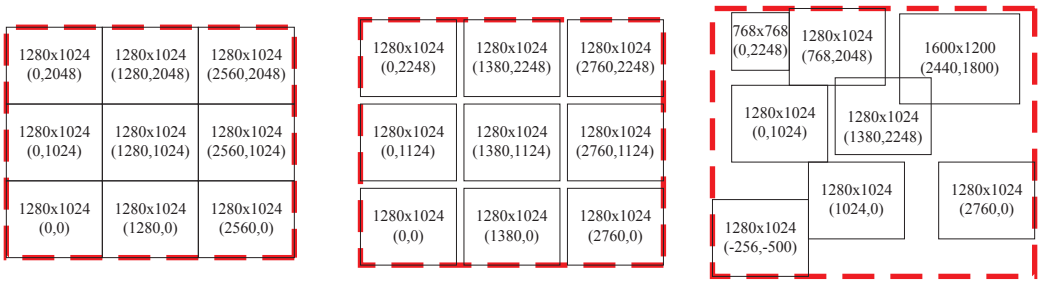


**Figure 3.1.** Defining a tile display with viewports in a logical global display. Three possible tile arrangements are shown. The bounds of each tile is drawn with the viewport given inside. The viewable area is shown with a dashed line.

IceT defines the configuration of a tiled display by using a **logical global display** with an infinite [1] number of pixels in both the horizontal and vertical directions. The definition of each tile comprises the identifier for the process connected to the physical projection and the viewport (position and size) of the tile in the global display. IceT implicitly defines the rectangle that tightly encompasses all of the tile viewports as the viewable area and snaps the viewing region (defined by the OpenGL viewing matrices) to this area.

Figure 3.1 shows some possible tile arrangements. Mullions or overlaps of the tiles in the physical display can be represented by the spacing or overlap of the viewports in the logical display. IceT does not require the tile layout to have any regularity. Chaotic layouts like that shown in the right image of Figure 3.1 are legal, although probably not very useful. It is allowed, and in fact encouraged, to have processes that are not directly connected to the tiled display. These **non-display** processes still contribute to the image compositing work and will reduce the overall time to render an image.

The display is defined using the **`icetResetTiles`** and **`icetAddTile`** functions. Any previous tile definition is first cleared out using **`icetResetTiles`** and new tiles are added, one at a time, using **`icetAddTile`**.

```
void icetResetTiles( void )

int icetAddTile( GLint    x,
                 GLint    y,
                 GLsizei  width,
                 GLsizei  height,
                 int      display_rank );
```

Each tile is specified using screen coordinates in the logical global display: the position of the

[1] Well, OK. The logical global display only stretches as far as the 32-bit numbers that are used to define viewports. But that's still way bigger than any physical display that we can possibly conceive, so conceptually we call it infinite.

lower left corner and the width and height of the tile. Each tile also has a **display process** associated with it. After an image is completely rendered and composited, the screen section belonging to this tile will be placed in the process at the given rank.

The following code demonstrates a common example for establishing the tile layout: a grid of projectors. The arrangement of projectors in this example assume that the projectors are connected to processes in the order of left to right and then top to bottom, which is common. Note, however, that IceT defines its logical global display with *y* values from the bottom up like OpenGL does.

```
icetResetTiles();
for (row = 0; row < num_tile_rows; row++) {
    for (column = 0; column < num_tile_columns; column++) {
        icetAddTile(column*TILE_WIDTH, (num_tile_rows-row-1)*TILE_HEIGHT,
                    TILE_WIDTH, TILE_HEIGHT,
                    row*num_tile_columns + column);
    }
}
```

**Mullions** are added by simply spacing the tiles apart from each other in the logical global display. Because they are defined in the logical global display, physical dimensions of the mullions must first be converted to pixels using the dot pitch of the displays. The following code adds mullions between all of the tiles.

```
icetResetTiles();
for (row = 0; row < num_tile_rows; row++) {
    for (column = 0; column < num_tile_columns; column++) {
        icetAddTile(column*(TILE_WIDTH + x_mullion),
                    (num_tile_rows-row-1)*(TILE_HEIGHT + y_mullion),
                    TILE_WIDTH, TILE_HEIGHT,
                    row*num_tile_columns + column);
    }
}
```

An equally common use for IceT is to render images in parallel to a single display. In this **single-tile rendering** mode, we simply create a single tile whose image will be placed in the GUI of some application. This is done by either using the OpenGL context of the GUI as part of the IceT rendering process or by grabbing the image of the single tile and copying into the GUI. The example code below sets up IceT to create a single image that is accessible on the root process.

```
icetResetTiles();
icetAddTile(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT, 0);
```

IceT indexes the tiles in the order that they are defined with **`icetAddTile`**. You can get the current definition of the tile display from a number of state variables, which can be retrieved

as always with **icetGet**. **ICET_NUM_TILES** stores the number of tiles that are defined (the number of times **icetAddTile** was called). **ICET_TILE_VIEWPORTS** stores an array with all of the dimensions of each tile. For each tile, **ICET_TILE_VIEWPORTS** contains the four values $\langle x, y, width, height \rangle$, stored consecutively, corresponding to the values passed to **icetAddTile**. **ICET_DISPLAY_NODES** stores an array giving the rank of the display process displaying that tile. Each process can also query the **ICET_TILE_DISPLAYED** variable to see which tile is displayed locally. **ICET_TILE_DISPLAYED** is set to $-1$ on every process that does not display a tile.

You can get information about the display geometry as a whole through **ICET_GLOBAL_VIEWPORT**. This variable stores the four-tuple $\langle x, y, width, height \rangle$. $x$ and $y$ are placed at the left-most and lowest position of all the tiles, and *width* and *height* are just big enough for the viewport to cover all tiles.

Calling **icetAddTile** will not create a display context for the tile. That responsibility is left to the calling application. IceT will use whatever OpenGL context is current. When using IceT in single-tile rendering mode, the OpenGL context should simply be set to the image size. When driving a physical tile display, each display process must create a window that covers the entire display. It is also a good idea to disable the mouse cursor in these windows.

Note that the size of the tiles do not have to match each other. Also, the size of the OpenGL viewport does not have to match the size of any of the tiles. There is, however, a constraint that the OpenGL viewport on all processes must be at least as large as the largest tile in each dimension. To help you maintain that constraint, IceT stores the largest tile dimensions in the **ICET_TILE_MAX_WIDTH** and **ICET_TILE_MAX_HEIGHT** state variables. The overall maximum tile size is provided in **ICET_TILE_MAX_PIXELS** so that you can allocate buffers big enough for any tile image.

Although counterintuitive, it is often more efficient to create OpenGL viewports that are larger than any tile. This situation may be necessary when using image inflation (see Chapter 4). Even when not using image inflation, larger rendering viewports can save a significant amount of rendering time. IceT can use the larger OpenGL image buffer to potentially render in one shot an object that is larger than any of the tiles.

# Strategies

IceT contains several algorithms for performing image compositing. The overall algorithm used to render and composite an image is called a strategy, named after the "Gang of Four" strategy pattern.[2] The strategy is set using the **icetStrategy** function.

```
void icetStrategy( IceTStrategy strategy );
```

IceT defines the following strategies that can be passed to **icetStrategy**. These strategies

[2]Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994. ISBN 0-201-63361-2.

are discussed in more detail in Chapter 5.

**ICET_STRATEGY_REDUCE** A two phase algorithm. In the first phase, tile images are redistributed such that each process has one image for one tile. In the second phase, a 'traditional' single tile composition is performed for each tile. Since each process contains an image for only one tile, all these compositions may happen simultaneously. This is a well rounded strategy that seems to perform well in a wide variety of applications.

**ICET_STRATEGY_SPLIT** Each tile is split up, and each process is assigned a piece of a tile in such a way that each process receives and handles about the same amount of data. This strategy is often very efficient, but due to the large amount of messages passed, it has not proven to be very scalable or robust.

**ICET_STRATEGY_VTREE** An extension to the binary tree algorithm for image composition. Sets up a "virtual" composition tree for each tile image. Processes that belong to multiple trees (because they render to more than one tile) are allowed to float between trees. This strategy is not quite as well load balanced as **ICET_STRATEGY_REDUCE** or **ICET_STRATEGY_SPLIT**, but has very well behaved network communication.

**ICET_STRATEGY_SERIAL** Basically applies a "traditional" single tile composition (such as binary swap) to each tile in the order they were defined. Because each process must take part in the composition of each tile regardless of whether they draw into it, this strategy is usually very inefficient when compositing for more than tile. It is provided mostly for comparative purposes.

**ICET_STRATEGY_DIRECT** As each process renders an image for a tile, that image is sent directly to the process that will display that tile. This usually results in a few processes receiving and processing the majority of the data, and is therefore usually an inefficient strategy.

# Drawing Callback

Most compositing engines will simply take a group of images and combine them together. This approach, however, is unreasonable when compositing the high resolution images on a large tiled display. It is problematic for an application to create images larger than any color buffer the rendering hardware can create, and holding many of these large images can lead to a large memory profile.

Instead, the IceT algorithms deal with pieces of the overall image. The image pieces are created on demand. As such, IceT may require the same geometry to be rendered multiple times in a single frame. IceT provides the application with the most flexible way to define the rendering process: with a **drawing callback**.

A drawing callback is simply a function that your application provides IceT. When IceT needs an image, it will establish the appropriate OpenGL transformation matrices for the section of the

overall display it needs. The application's drawing callback will issue the appropriate OpenGL commands to draw the geometry. The drawing callback leaves its image in the OpenGL buffers upon exiting.

The drawing callback is set with the **icetDrawFunc**.

```
typedef void (*IceTCallback)(void);

void icetDrawFunc( IceTCallback func );
```

IceT can nominally call the drawing callback for every tile in the display. However, in almost any real application each process has data that demonstrates some spatial locality that causes it to be projected on a relatively small section of the display. To give IceT the information it needs to prevent unnecessary renders, the application needs to provide the bounds of the local geometry. This is done using either the **icetBoundingVertices** or the **icetBoundingBox** function.

```
void icetBoundingVertices( GLint           size,
                           GLenum          type,
                           GLsizei         stride,
                           GLsizei         count,
                           const GLvoid *  pointer );

void icetBoundingBoxd ( GLdouble  x_min,
                        GLdouble  x_max,
                        GLdouble  y_min,
                        GLdouble  y_max,
                        GLdouble  z_min,
                        GLdouble  z_max );

void icetBoundingBoxf ( GLfloat  x_min,
                        GLfloat  x_max,
                        GLfloat  y_min,
                        GLfloat  y_max,
                        GLfloat  z_min,
                        GLfloat  z_max );
```

With the **icetBoundingVertices** function, you specify a set of vertices whose convex hull completely contains the geometry. The **icetBoundingBox** function is a convenience function that defines the container as an axis aligned bounding box.

The drawing callback should *not* modify the **GL_PROJECTION_MATRIX** as this would cause IceT to place image data in the wrong location in the tiled display and improperly cull geometry. It is acceptable to add transformations to **GL_MODELVIEW_MATRIX**, but the bounding vertices given with **icetBoundingVertices** or **icetBoundingBox** are assumed to already be transformed by any such changes to the modelview matrix. Also, **GL_MODELVIEW_MATRIX** must be restored before the draw function returns. Therefore, any changes to **GL_MODELVIEW_MATRIX** are to be done with care and should be surrounded by a pair of glPushMatrix and glPopMatrix

functions.

It is also important that the drawing callback *not* attempt the change the clear color. In some compositing modes, IceT needs to read, modify, and change the background color. These operations will be lost if the drawing callback changes the background color, and severe color blending artifacts may result.

IceT may call the drawing callback several times to create a single tiled image or not at all if the current bounds lie outside the current view frustum. This can have a subtle but important impact on the behavior of the drawing callback. For example, counting frames by incrementing a frame counter in the drawing callback is obviously wrong (although you could count how many times a render occurs). The drawing callback should also leave OpenGL in a state such that it will be correct for a subsequent run of the drawing callback. Any matrices or attributes pushed in the drawing callback should be popped before the drawing callback returns, and any state that is assumed to be true on entrance to the drawing callback should also be true on return.

# Rendering

Once you have set up the IceT state as described in the previous sections of this chapter, you are ready to perform parallel rendering. Parallel rendering is performed by calling **icetDrawFrame**.

```
void icetDrawFrame(void);
```

**icetDrawFrame** is called in basically the same way as the drawing callback would be called directly. First, establish the OpenGL state. Setting up the **GL_PROJECTION_MATRIX** before calling **icetDrawFrame** is essential. It is also advisable to set up whatever transformations in the **GL_MODELVIEW_MATRIX** that you can before calling **icetDrawFrame**. IceT will use and modify these two matrices to render regions of the tiled display. The drawing callback should behave as if neither of the matrices were modified.

By the time **icetDrawFrame** completes, an image will have been completely rendered and composited. If **ICET_DISPLAY** is enabled, then the fully composited image is written back to the OpenGL framebuffer for display. It is the application's responsibility to synchronize the processes and swap front and back buffers. The image remaining in the frame buffer is undefined if **ICET_DISPLAY** is disabled or the process is not displaying a tile.

If the OpenGL background color is set to something other than black, **ICET_DISPLAY_COLORED_BACKGROUND** should also be enabled. Displaying with **ICET_DISPLAY_COLORED_BACKGROUND** disabled may be slightly faster (depending on graphics hardware) but can result in black rectangles in the background.

If **ICET_DISPLAY_INFLATE** is enabled and the size of the renderable window (determined by the current OpenGL viewport) is greater than that of the tile being displayed, then the image will first be "inflated" to the size of the actual display. If **ICET_DISPLAY_INFLATE** is disabled,

the image is drawn at its original resolution at the lower left corner of the display. More details on image inflation are given in Chapter 4.

Regardless of whether it writes the fully composited image back to the display, IceT stores the resulting image buffers. These image buffers can be retrieved with the **icetGetColorBuffer** and **icetGetDepthBuffer** functions.

```
GLubyte  *icetGetColorBuffer  ( void );
GLuint   *icetGetDepthBuffer  ( void );
```

Of course, color buffers are only available on display processes. Also be aware that a color or depth buffer may not have been computed with the last call to **icetDrawFrame**. IceT avoids the computation and network transfers for any unnecessary buffers unless specifically requested otherwise with the flags given to the **icetInputOutputBuffers** function.

```
void icetInputOutputBuffers( GLenum inputs,
                             GLenum outputs );
```

**icetInputOutputBuffers** allows you to specify which OpenGL buffers to composite (the *input* buffers) and which to deliver to the display processes (the *output* buffers). The color and depth buffers are specified with the **ICET_COLOR_BUFFER_BIT** and **ICET_DEPTH_BUFFER_BIT**, respectively. By default, IceT reads in both the color and depth buffers, performs compositing using **Z comparison**, and delivers just the color buffer to the display nodes. If you need the depth buffer as well, specify depth as one of the outputs. If you do not need the color buffer, you can remove the color buffer as one of the input buffers. If the depth buffer is not specified as one of the inputs, then the compositing mode is automatically switched to **alpha blending**. However, alpha blending requires additional information from the application, which is discussed in Chapter 4.

In any case, you can query whether a color or depth buffer is available with **ICET_COLOR_BUFFER_VALID** or **ICET_DEPTH_BUFFER_VALID**, respectively. It is an error to read an invalid buffer.

The memory returned by **icetGetColorBuffer** and **icetGetDepthBuffer** need not, and should not, be freed. It will be reclaimed in the next call to **icetDrawFrame**. Expect the data returned to be obliterated on the next call to **icetDrawFrame**.

# Chapter 4

# Customizing Compositing

If you have been reading this document from the beginning, then you already know enough to use IceT for many typical rendering applications. Chapters 2 and 3 describe how to build and link IceT, establish an IceT context in your application, and to leverage IceT to make your rendering parallel. This chapter describes the many features IceT provides to let you customize the image compositing to your application.

## Compositing Operation

IceT is classified as a **sort-last** type of parallel rendering library, as discussed in Chapter 1. Basically, this means that each process renders images independently, and then these images, each comprising a different partition of the geometry, are combined together in a process called **compositing**.

To combine two images together, a **compositing operation** is applied to every corresponding pair of pixels. Three or more images are combined by applying the compositing operation multiple times to eventually reduce everything to one image. (The compositing operations supported by IceT are associative, so order does not matter. IceT takes advantage of this fact to efficiently perform the compositing in parallel.)

IceT supports two compositing operations. The first type of compositing operation is a depth comparison and the other is an alpha blend. The depth comparison is a bit faster and is easier to use, but only works for opaque surfaces. If you are performing **volume rendering**, the translucent rendering of 3-dimensional volumes, or any other rendering that involves transparent data, then you will have to use the alpha blend compositing operation.

### Z-Buffer Compositing

**Z-buffer compositing** takes advantage of the same hidden surface removal already taking place in the OpenGL pipeline. IceT pulls the z-buffer (also often known as the **depth buffer**) from the OpenGL image buffers. The compositing operation then just compares the depth values of two pixels and chooses the one that is closer.

Z-buffer compositing is used whenever the depth buffer is chosen as one of the input buffers. The input (and output) buffers are chosen with the **icetInputOutputBuffers** function.

```
void icetInputOutputBuffers( GLenum inputs,
                             GLenum outputs );
```

By default, both the the color and the depth buffer are selected as input buffers and the color buffer is selected as the only output buffer. This means that the depth buffer will be used to do z-buffer compositing, but only the color buffer will be fully composited. (Not computing the depth buffer may save some network transfer time.)

If you need the depth buffer composited in addition to the color buffer (for example, to help with a picking operation), you can do so by simply setting the depth buffer as one of the output buffers.

```
icetInputOutputBuffers(ICET_COLOR_BUFFER_BIT | ICET_DEPTH_BUFFER_BIT,
                       ICET_COLOR_BUFFER_BIT | ICET_DEPTH_BUFFER_BIT);
```

Alternatively, if you only need the depth buffer (for example, as a shadow map), you can do so by setting both the input and output buffers to just the depth buffer.

```
icetInputOutputBuffers(ICET_DEPTH_BUFFER_BIT, ICET_DEPTH_BUFFER_BIT);
```

## VOLUME RENDERING (AND OTHER TRANSPARENT OBJECTS)

A well known limitation to z-buffer compositing — and the z-buffer hidden surface removal algorithm in general — is that it only works with opaque objects. You will get invalid results if you try to apply z-buffer compositing on transparent objects.

There are two fundamental problems with the z-buffer compositing operation when dealing with translucent pixels. The first problem is that you cannot simply pick the nearest color value. You must **blend** the front pixel's color with the back pixel's color. The second problem is that the color blending is order dependent. That is, you have to know which pixels are in front of others. Although it is technically possible to use z-buffer values to determine the ordering of a pair of pixels, making sure that all the pixels get composited in the correct order requires additional information about and constraints on the geometry.

When z-buffer compositing is not applicable, you must use **blended compositing**. Blended compositing is automatically turned on when there is no z-buffer specified as an input buffer. That generally means you will be setting both the input and output buffers to the color buffer.

```
icetInputOutputBuffers(ICET_COLOR_BUFFER_BIT, ICET_COLOR_BUFFER_BIT);
```

The blending composite operator relies on the **alpha** ($\alpha$) channel of the color buffer (the A in RGBA colors). Note that the alpha values must actually be available in the OpenGL color buffers in order for blended compositing to work. Many applications create OpenGL buffers without alpha bit planes in them because they are often not necessary to render images in serial. Make sure your application creates alpha bit planes before attempting to composite translucent images with IceT (or any other library).

The blending operation is the standard **over/under operator** defined in the seminal 1984 Porter and Duff paper.

$$C_o \leftarrow C_f + C_b(1 - \alpha_f) \tag{4.1}$$

where $C$ is an RGBA color vector, $\alpha$ is the alpha component of a color vector, and the $f$, $b$, and $o$ subscripts denote the front, back, and output values, respectively.

Each color in Equation 4.1 represents a **pre-multiplied color**, meaning that the red, green, and blue values are scaled by the alpha parameter. Thus, a fully red color at half transparency is represented by the vector $\langle 0.5, 0, 0, 0.5 \rangle$ rather than $\langle 1, 0, 0, 0.5 \rangle$. In pre-multiplied colors, none of the red, green, or blue values ever exceed the alpha value. Note that colors are often provided in OpenGL as non-pre-multiplied values, and the blending equation $C_o \leftarrow C_f \alpha_f + C_b(1 - \alpha_f)$ is used instead of the one in Equation 4.1. Although this blending gives the correct RGB color, it computes an invalid alpha parameter, so watch out!

Simply turning on blended compositing is not sufficient to render translucent objects. You must also tell IceT to perform **ordered compositing**. In ordered compositing, you must have a **visibility ordering**. Given any two processes, a visibility ordering ensures and determines that all of the geometry in one process is in front of or behind all the geometry in each of the other process with respect to the camera. In some cases, such as when volume rendering a 3D Cartesian grid of points distributed in blocks to processes, finding the visibility ordering is straightforward. In other cases, such as when rendering unstructured collections of polygons or polyhedra, it can be difficult to ensure that a visibility ordering exists and can be found. Doing so may be the most challenging part of creating a parallel rendering application. An example of creating a visibility ordering from unstructured data can be found in the ParaView application, and the implementation is detailed in the following paper:

> Kenneth Moreland, Lisa Avila, and Lee Ann Fisk. "Parallel Unstructured Volume Rendering in ParaView," In *Visualization and Data Analysis 2007, Proceedings of SPIE-IS&T Electronic Imaging*, January 2007, pp. 64950F-1–12.

Ordered compositing is turned on by simply passing the **`ICET_ORDERED_COMPOSITE`** flag to **`icetEnable`**.

```
icetEnable(ICET_ORDERED_COMPOSITE);
```

Once ordered compositing is enabled, it is very important to use **icetCompositeOrder** to specify the visibility order of the geometry associated with each process. This must generally be done before each call to **icetDrawFrame**.

```
void icetCompositeOrder( const GLint * process_ranks );
```

The **icetCompositeOrder** function takes an array of processes. It is assumed that the geometry of the first process in the list is in front of the rest of the processes; the geometry of the second process in the list is in front of all the processes except the first, and so on. The visibility order often changes when the camera angle changes, so it is important to recompute and report a new composite order on every frame.

Be aware that not all strategies support ordered compositing. If the current strategy does not support ordered compositing, then the **ICET_ORDERED_COMPOSITE** flag is ignored. Consult the documentation in Chapter 5 or the documentation for the **icetStrategy** command to determine which strategies support ordered compositing. In any case, you can check the **ICET_STRATEGY_SUPPORTS_ORDERING** state variable to determine if the current compositing strategy supports ordered compositing.

One final thing to worry about when using blended compositing is to make sure that the background color does not interfere with the compositing. Because the visibility order is important, you need to make sure that none of the processes render with a background (except perhaps the process nearest the rear). For example, let us say you want to render an image with a blue background. Let us also say that process $A$'s geometry is in front of process $B$'s geometry. Process $A$ cannot render its geometry on top of a blue background because that background should really also be behind the geometry of process $B$, and the resulting image will be invalid.

If your background is a solid color, then IceT can fix this problem automatically. Simply set the OpenGL background (clear) color like you normally would and enable the **ICET_CORRECT_COLORED_BACKGROUND** feature.

```
glClearColor(0.0, 0.0, 1.0, 1.0);
icetEnable(ICET_CORRECT_COLORED_BACKGROUND)
```

When the **ICET_CORRECT_COLORED_BACKGROUND** feature is enabled and blended compositing is on, IceT will change the background to $\langle 0,0,0,0 \rangle$, perform the rendering and compositing, blend the result into the specified background color, and finally restore the OpenGL clear color.

If you do not actually need to get the image result back from **icetGetColorBuffer**, you can use the **ICET_DISPLAY_COLORED_BACKGROUND**.

```
glClearColor(0.0, 0.0, 1.0, 1.0);
icetEnable(ICET_CORRECT_COLORED_BACKGROUND)
```

**ICET_DISPLAY_COLORED_BACKGROUND** operates similar to **ICET_CORRECT_COLORED_BACKGROUND** with the exception that it uses the OpenGL graphics hardware to blend the composited image to the colored background, and may therefore get a modest performance increase. However, it also means that the result will not be available in the memory buffer returned by **icetGetColorBuffer**.

# Image Inflation

Because IceT is an image-based sort-last parallel rendering library, its overhead is proportional to the size of the images being generated. Thus, large displays can limit the maximum rendering frame rate that can be achieved.

A simple way to increase the frame rate is to reduce the resolution of the images being displayed. If the display resolution is larger than necessary (and "larger than necessary" is a flexible metric that can change regularly as an application runs), then you can tell IceT to render smaller images and then **inflate** the images to fill the display. A major use case for a reduced resolution image is for maintaining application interactivity. Many applications, particularly visualization applications, contain bursts of interactivity. The user will interact with the data (move the camera or objects) and then hold still and analyse the results. While interacting, application responsiveness is much more important than image details, so during this time a lower resolution image can be rendered and inflated. When the user stops interacting and starts analysing, a full resolution image can be created.

You can instruct IceT to render and composite smaller images by simply specifying a lower resolution display with the **icetAddTile** function. If you are frequently switching the resolution of the images being generated (which is common), then you can use IceT state management to switch states. First, use **icetCreateContext** and **icetCopyState** to create a duplicate state. Then change the display of one of the states to a lower resolution with **icetAddTile**. As the application runs, use **icetSetContext** to swap between the different resolutions. See Chapter 3 for details on using these functions.

Between rendering and display, the smaller images must be inflated to fill the display. An application can always perform this inflation itself (and that is probably necessary if the images are shipped to a remote display). When IceT is displaying the data (i.e. **ICET_DISPLAY** is enabled), IceT has the ability to automatically inflate the images. Turn on this feature by enabling **ICET_DISPLAY_INFLATE**. IceT contains two modes for inflating images: using the CPU or using texture mapping in OpenGL. When **ICET_DISPLAY_INFLATE_WITH_HARDWARE** is enabled (the default), then texture mapping is used. In either case, **icetGetColorBuffer** and **icetGetDepthBuffer** return the smaller image size specified by **icetAddTile**.

One final note: Regardless of what size you set for the displays in **icetAddTile**, you should keep the viewport (specified by **glViewport**) as large as possible. The size of the graphics viewport and the size of the tile images can be different so long as each viewport is at least as large

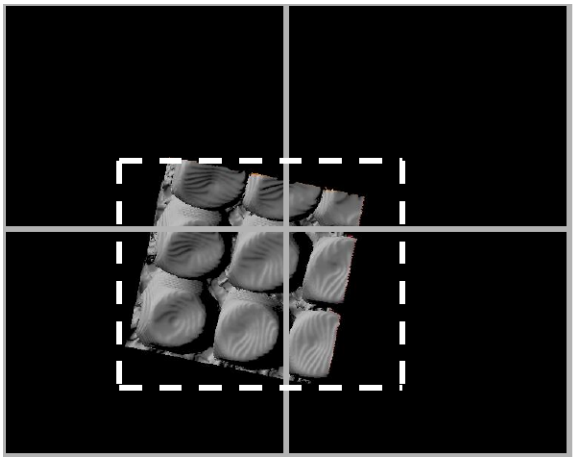

**Figure 4.1.** Even though geometry may straddle tile boundaries, we may be able to render it all in one pass by "floating" the viewport.

as the largest tile image. In fact, it is advantageous to have the viewport larger than the specified tiles. The first reason is that the **ICET_DISPLAY_INFLATE** feature fills the image to the OpenGL viewport. If the dimensions the two are the same, then no inflation will actually take place. The second reason is that IceT will use the entire OpenGL viewport for rendering. For a multi-tile display, this can dramatically reduce the number of times the render callback needs to be called. Thus, in general it is best to keep the OpenGL viewport as large as possible.

# Floating Viewport

Consider the geometry shown in Figure 4.1 that projects onto a screen space that fits within a single tile but is moved in the horizontal and vertical directions so that it straddles four tiles. If the system limits itself to projecting onto physical tiles, the processor must render and read back four images; although it could generate a single image that contains the entire geometry with the exact same pixel spacing. Instead of rendering four tiles, the system can **float** the viewport in the global display to the space straddling the tiles. That is, the system may project the geometry to the space shown by the dotted line in Figure 4.1 and split the resulting image back into pieces that can be displayed directly on each tile. Hence, the system does not need to render any polygon more than once, and the frame buffer is read back one time instead of four.

When a processors geometry fits within the floating viewport, it can cut the rendering time dramatically. This is most likely to happen when the number of tiles is small compared to the number of processors and the spatial coherency of the data is good.

The floating viewport is always enabled by default. You can disable it by calling **icetDis-**

**able** with the **ICET_FLOATING_VIEWPORT** identifier. In general, there is not much reason to turn off the floating viewport. The only real reason to turn off the floating viewport is to prevent IceT from changing the perspective matrix when in single tile mode. However, IceT will change the perspective matrix anyway when rendering with more than one tile, so any application that might render to a tiled display should simply leave the floating viewport option on.

# Active-Pixel Encoding

Because each processor renders only a fraction of the total geometry, the geometry often occupies only a fraction of the screen space in some or all of the tiles in which it lies. Consequently, the initial images distributed between processors at the beginning of composition often have a significant amount of blank space within them. Explicitly sending this information between processors is a waste of bandwidth. Transferring sparse image data rather than full image data is a well-known way to reduce network overhead. So far, our best method to do this has been with active-pixel encoding.

Active-pixel encoding is a form of run-length encoding. A traditional run-length encoding groups pixels into contiguous groups where the color and/or depth does not change. However, in a practical 3D rendering, both the color and depth change almost everywhere except in the background areas where nothing is rendered. To take advantage of this, images are grouped into alternating run lengths of **active pixels**, pixels that contain geometry information, and **inactive pixels**, pixels that have no geometry drawn on them. The active-pixel run length is followed by pairs of color and depth values (or just one of the two if that is the only data available).. The inactive pixels are not accompanied by any color or depth information. The depth information is assumed to be of maximum depth, and the color values are ignored since they contain no geometry information.

There are many other ways to encode sparse images and reduce data redundancy. However, we are particularly enamored with our active-pixel encoding for this application because it exhibits all of the following properties:

**Fast encoding** Image encoding requires each pixel to be visited exactly once. Each visit includes a single depth buffer comparison, a single addition, and at most one copy.

**Free decoding** Processors typically perform a depth comparison as soon as they receive incoming data. The depth comparison can be done directly against an image that is still encoded in sparse form. In fact, the depth comparison can skip the comparisons for the inactive pixels. Thus doing depth comparisons against encoded images is often faster than against unencoded images.

**Effective compression** During the early stages of composition when the largest images must be transferred, the sparse data is commonly less than one fifth the size of the original data.

**Good worst case behavior** No image with both color and depth information (the most common case) will ever grow by more than a few bytes of header information. Images that have geometry drawn on every pixel will only have one run length. Even images that alternate between active and inactive status for every pixel, and hence have a run length for every pixel, do not grow when encoded. The number of bytes required to record two run lengths is equal to the number of bytes saved by not recording color and depth information for a single inactive pixel. Thus, there is no penalty for recording run lengths of size one. If only color or only depth is being recorded, it is possible to grow data in the pathological case where pixels alternate between active and inactive, but in practice background pixels are grouped and the data never really grows.

Active-pixel encoding is performed automatically during the compositing process. There is currently no way to turn it off.

# Data Replication

The primary advantage of IceT's parallel rendering algorithms, and sort-last rendering algorithms in general, is that they are very scalable with respect to the size of the input geometry. That is, there is no overhead to adding more geometry other than the time it takes hardware to render and there is only a logarithmic overhead for adding processors to the job.

The down side of a sort-last approach is that the image compositing overhead must be incurred regardless of how little geometry is being rendered. This overhead limits the maximum frame rate that can be achieved by the parallel rendering. Consequently, the parallel rendering can potentially be slower than the serial rendering if the amount of geometry being rendered is small.

One possible way to get higher frame rates with smaller geometries would be to switch to a different parallel rendering mode, but doing so is unnecessarily complicated. Another possibility is to collect the data on a single process and circumvent the IceT library entirely. This approach is fine when using single tile mode where the image is displayed at a single location, but is not at all straightforward when displaying to a tiled display.

IceT provides a better solution than either of the previous two approaches. If the image compositing work is dominating the rendering time, you can set up a **data replication group**. To set up a data replication group, you partition the geometry using fewer partitions than processes, and then you share each partition with multiple processes. The processes that share a data partition are a replication group. IceT will divide the compositing work for each replication group amongst the processes in the group. In essence, you are adding geometry rendering work to remove image compositing work.

One of the most common uses for data replication groups is to simply replicate the same geometry on all processes. This is helpful, for example, if your application supports lower levels of detail for interaction. The lower level of detail can be replicated on all processes. However,

you could also conceivably arrange for any amount of replication between all replicated and none replicated for a consistently appropriate overhead as the amount of data grows.

To set up data replication groups, it is your responsibility to partition and replicate geometry. (IceT knows nothing about geometry.) You then report what the data replication groups are with the **icetDataReplicationGroup** function.

```
void icetDataReplicationGroup( GLint          size,
                               const GLint *  processes  );
```

**icetDataReplicationGroup** simply takes an array defining the replication group that the local process belongs to. It is important to ensure that all processes belonging to a group provide the same array to **icetDataReplicationGroup**. As a convenience, IceT also provides the **icetDataReplicationGroupColor** function that allows you to define the data replication groups by assigning an identifier (i.e. color) to each partition and having each process report the partition color in which it belongs.

```
void icetDataReplicationGroupColor( GLint  color  );
```

As an example, let us say that processes 0–3 share the same geometry, 4–7 share the same geometry, 8–11 share the same geometry, and so on. These replication groups could be reported with the following call (where `rank` is the local process id as stored in the **ICET_RANK** state variable).

```
icetDataReplicationGroupColor(rank/4);
```

The data replication group is stored in the **ICET_DATA_REPLICATION_GROUP** state variable (retrievable with **icetGet**). The length of the group array is stored in **ICET_DATA_REPLICATION_GROUP_SIZE**. The data replication group array is available regardless of whether you used **icetDataReplicationGroup** or **icetDataReplicationGroupColor** to define the group. The default value is an array with one value: the local process.

# Timing (and Other Metrics)

Whenever **icetDrawFrame** is called, IceT measures the amount of time spent in the various tasks required for parallel rendering. These timings are stored in the IceT state and can be retrieved with **icetGet**. The state variables containing these timing metrics (in seconds) are as follows.

**ICET_RENDER_TIME** The total time spent in the drawing callback during the last call to **icetDrawFrame**.

**ICET_BUFFER_READ_TIME** The total time spent reading from OpenGL buffers during the last call to **icetDrawFrame**.

**ICET_BUFFER_WRITE_TIME** The total time spent writing to OpenGL buffers during the last call to **icetDrawFrame**.

**ICET_COMPRESS_TIME** The total time spent in compressing image data using active pixel encoding during the last call to **icetDrawFrame**.

**ICET_BLEND_TIME /ICET_COMPARE_TIME** The total time spent in performing Z comparisons or color blending of images during the last call to **icetDrawFrame**. These two variables always return the same value.

**ICET_RENDER_TIME** The total time spent in the drawing callback during the last call to **icetDrawFrame**.

**ICET_COMPOSITE_TIME** The total time spent in compositing during the last call to **icetDrawFrame**. Equal to **ICET_TOTAL_DRAW_TIME** − **ICET_RENDER_TIME** − **ICET_BUFFER_READ_TIME** − **ICET_BUFFER_WRITE_TIME**.

In addition to timing how long rendering and compositing takes, IceT also keeps track of how much data is transmitted during compositing. The state variable **ICET_BYTES_SENT** stores the total number of bytes sent by the calling process for transferring image data during the last call to **icetDrawFrame**. Obviously, each process could have a different value for **ICET_BYTES_SENT**.

IceT also keeps track of the number of times **icetDrawFrame** has been called. This number is stored in **ICET_FRAME_COUNT**.

# Chapter 5

# Strategies

IceT contains several parallel image compositing algorithms. The type of compositing algorithm to use is selected by choosing a **strategy**. This chapter describes the underlying algorithm of each strategy. This user's guide will give qualitative comparisons between the strategies, but for a more quantitative analysis, see the following paper.

> Kenneth Moreland, Brian Wylie, and Constantine Pavlakos. "Sort-last parallel rendering for viewing extremely large data sets on tile displays," In *Proceedings of IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, October 2001, pp. 85–154.

A strategy is specified using the **icetStrategy** function.

```
void icetStrategy( IceTStrategy strategy );
```

The *strategy* is set to one of the identifiers for the strategies documented in the following sections of this chapter. A string documenting the current strategy can be retrieved with the **icetGetStrategyName** function.

To describe the IceT compositing algorithms, we will use the example parallel rendering problem shown in Figure 5.1 where 6 processes are each rendering their own piece of a shuttle model to a two tile display.

In this example, processes are denoted, in no particular order, by the colors gray, red, blue, green, purple, and orange. The colors of the geometry correspond to the process that generated each piece. Image boarder colors denote the process that generates and holds that image. (Apologies to those having troubles resolving the colors due to poor display or vision deficiencies. It should not be hard to follow the descriptions either way.)

## Single Image Compositing

Before discussing the multi-tile image compositing algorithms implemented by IceT, we visit the standard single image compositing algorithms. You cannot directly select a single image composit-

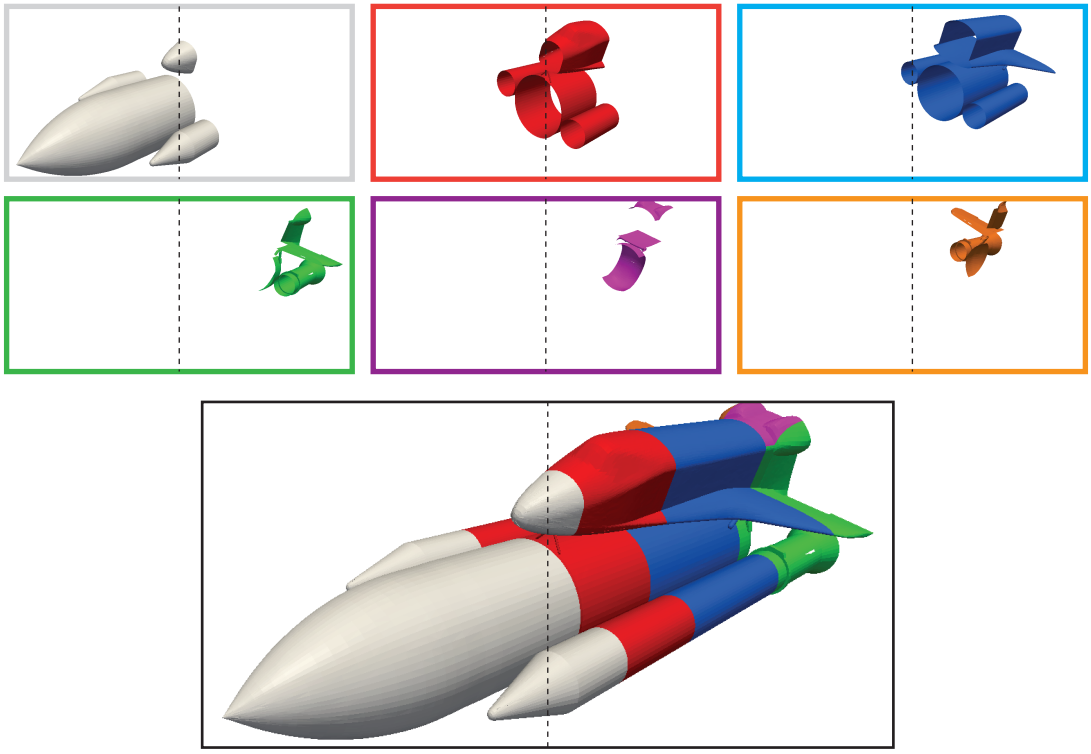

**Figure 5.1.** An example of six processes rendering to two tiles (top) and their composited image (bottom).

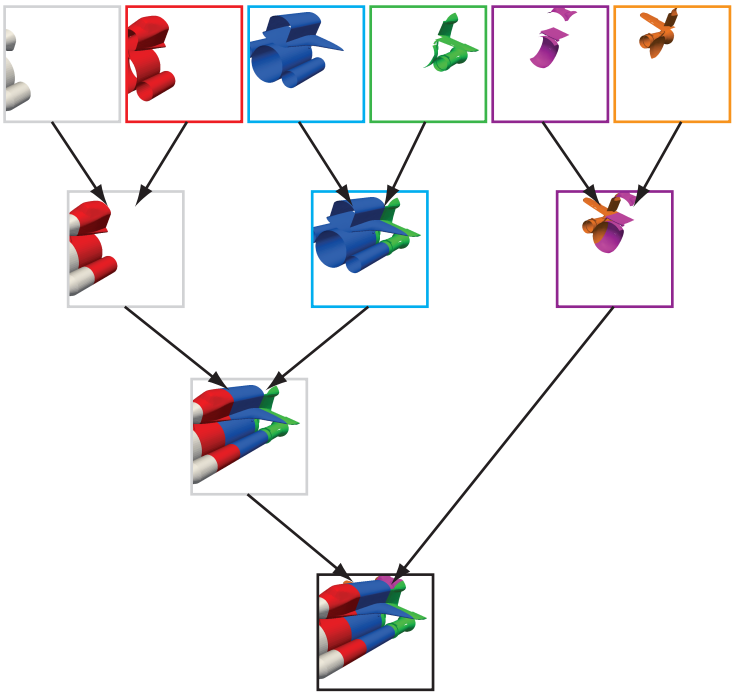

**Figure 5.2.** Tree composite network. Arrows represent the passing of data from one stage to the next. Processes receiving multiple images composite them together.

ing algorithm as a strategy (most of the multi-tile algorithms work well in "single-tile" mode), but these compositing algorithms are used as "subroutines" in some of the multi-tile algorithms. A reference to a **single image composite network** in the subsequent compositing algorithm descriptions refers to the algorithms described here.

## TREE COMPOSITING

IceT internally has two single image composite implementations. The first of which is the **tree composite** algorithm (sometimes also called binary tree composite due to its pair-wise grouping). The basic network for tree composite is shown in Figure 5.2.

The tree compositing algorithm is organized in stages. At each stage the processes pair up. One of the processes sends its data to its pair and then drops out of the computation. The receiving process combines the two images (using the compositing operation described in Chapter 4) and continues to the next stage. Processing continues until there is only one image (and one process) remaining.

As just defined, the tree composite algorithm only handles process counts that are a power of

two (that is, the number of processes is equal to $2^i$ for some integer $i$). IceT handles non-powers of two gracefully. At any stage where the number of processes is not even and one of the processes cannot be paired, that leftover process does nothing for that stage but then continues to participate in the next stage. An example of this can be seen in the second stage of Figure 5.2.

The advantages of tree composite are its regular and efficient data transfers. The limiting factor of tree compositing is that at each stage of the algorithm half of the processes drop out of the computation. Thus, for more than a few processes tree compositing provides poor process utilization.

## BINARY-SWAP COMPOSITING

The second single image compositing algorithm used internally by IceT is the **binary swap** algorithm. The basic network for binary-swap composite is shown in Figure 5.3

Like tree compositing, binary swap is organized in stages, and at each stage the processes pair up. However, rather than have one process send all the data to the other, the image space is divided in two and the processes swap image data so that each process has all the data for part of the image. At the next stage, the processes pair up again, but with different partners that have the same partition of the image. Processing continues until each of the $N$ processes have an image $1/N$ the size of the original image. At this point, all the processes send their sub-image to the display processes where the images are stitched together.

As just defined, the binary-swap composite algorithm only handles processes that are a power of two (that is, the number of processes is equal to $2^i$ for some integer $i$). Some binary-swap implementations handle non-powers of two by reducing the problem to the next largest power of two and dropping the leftover processes, but IceT handles non-powers of two more gracefully than that. Instead, IceT first finds the largest group of processes that is a power of two, makes a partition out of them, then finds the next largest group of processes that remain that is a power of two, makes a partition out of them, and so on. Each partition runs binary-swap independently up to the point where each process has its own piece of data. At this point, the smaller partitions send their image data to processes of the larger partitions, dividing up images where necessary. The largest partition then finishes the compositing in the normal way by collecting all of the pieces.

An example of compositing with a non-power of two is given in Figure 5.3. The six processes are partitioned first into a group of 4 and then into a group of 2. After swapping, the processes in the smaller group send images to the larger group. In this case, the purple process sends image data to the gray and blue processes, and the orange process sends to the red and green processes.

Like tree composite, binary swap exhibits regular and efficient data transfers. In addition, binary swap involves the use of all the processes throughout most of the compositing. Consequently, binary swap exhibits very good process utilization and scaling with respect to the number of processes on which it is run.

The most inefficient part of binary swap is the collection of image fragments at the end, which

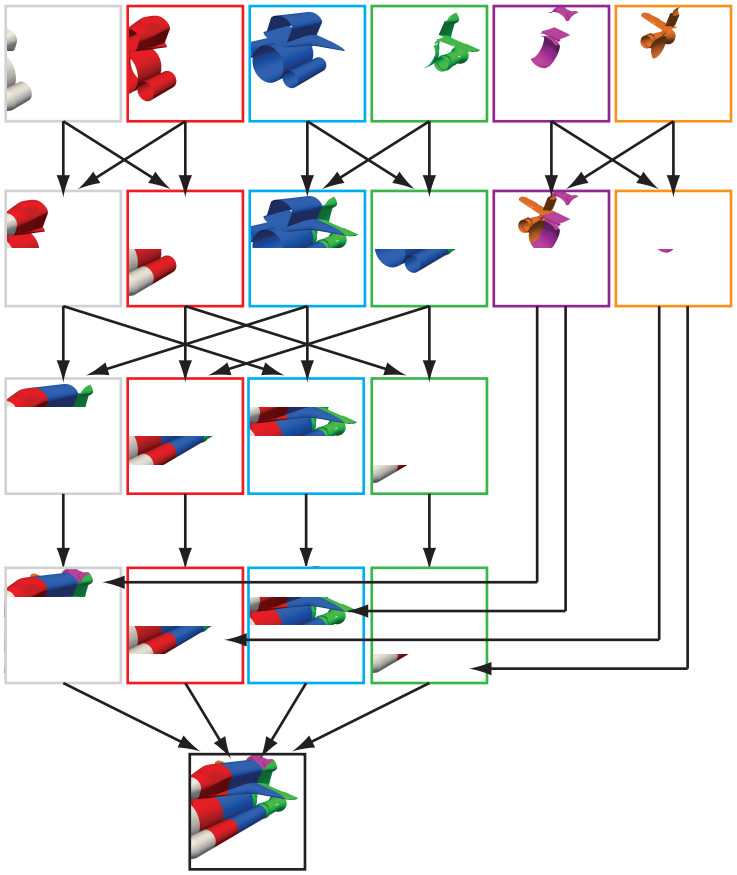

**Figure 5.3.** Binary-swap composite network. Arrows represent the passing of data from one stage to the next. Processes receiving multiple images composite or stitch them together. At most stages each process divides its image data and distributes it. The distribution of image data can be inferred from the target images.

is an extra step that tree composite does not need to take. Through some empirical studies, we found that the binary tree algorithm was more efficient than binary swap on less then 8 processes and less efficient on more than 8 processes. Consequently, IceT automatically switches between the two algorithms based on the amount of processes involved in the compositing.

### ORDERED COMPOSITING

In some applications, the order in which images are composited together makes a difference (see the Volume Rendering section in Chapter 4). The details on how ordered compositing is achieved is not given here, but the basic idea for both compositing algorithms is that they first swizzle the processes so that their order matches the order in which the images need to be composited together. When compositing images together, they make sure to maintain over/under constancy based on the swizzled ranks from the originating processes. The networks are also managed such that no two images are composited that are not directly "next" to each other (that is, there is no image that needs to be inserted between them).

## Reduce Strategy

An effective strategy implemented in IceT is the **reduce to single tile strategy** (or simply the reduce strategy). In this strategy, the multi-tile composite problem is efficiently reduced to a set of single image compositing problems, which are well studied and discussed in the previous section. The reduce strategy is selected by calling **`icetStrategy`** with the **`ICET_STRATEGY_REDUCE`** argument.

The reduce strategy is performed in two phases. In the first phase, processes are partitioned into groups, each of which is responsible for compositing the image of one of the tiles. The number of processes assigned to each tile is proportional to the number of non-empty images rendered for the corresponding tile. In the example shown in Figure 5.4 there are a total of 9 non-empty images. The left tile has 3 of the 9, that is $\frac{1}{3}$, of the images and thus is assigned $\frac{1}{3} \times 6 = 2$ processes. Likewise, the right image is assigned $\frac{2}{3} \times 6 = 4$ processes.

When assigned processes to tiles, display processes and processes rendering images to the tile are given preference. In the example of Figure 5.4, the gray and blue processes are assigned to the left tile. The remainder are assigned to the right tile. Any image generated by a process that does not belong to the destination tile is transferred to a process assigned to the tile. In the example, the three processes that render two images, gray, red, and blue, each pass one of their images to a process in the opposing process group. All of these transfers have unique senders and receivers and thus can happen simultaneously.

In the second phase of the reduce strategy, each group of processes independently composites its images together using one of the single image compositing algorithms described in the preceding section.

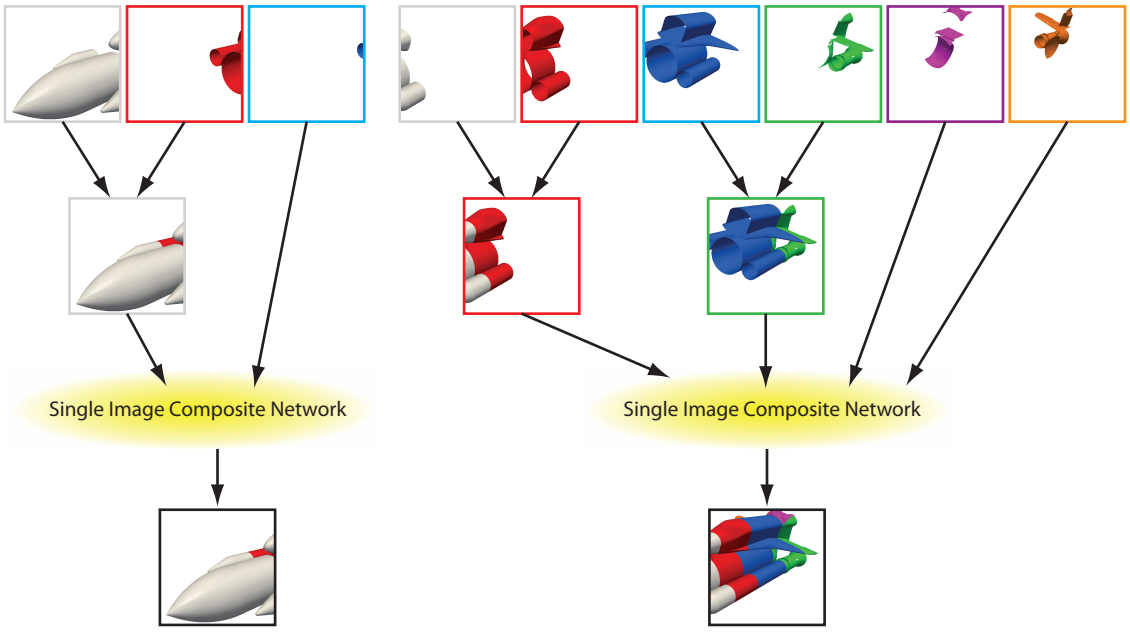


**Figure 5.4.** Composite network for reduce strategy. Arrows represent the passing of data from one stage to the next. Processes receiving multiple images composite them together. The single image composite network is described in a preceding section.

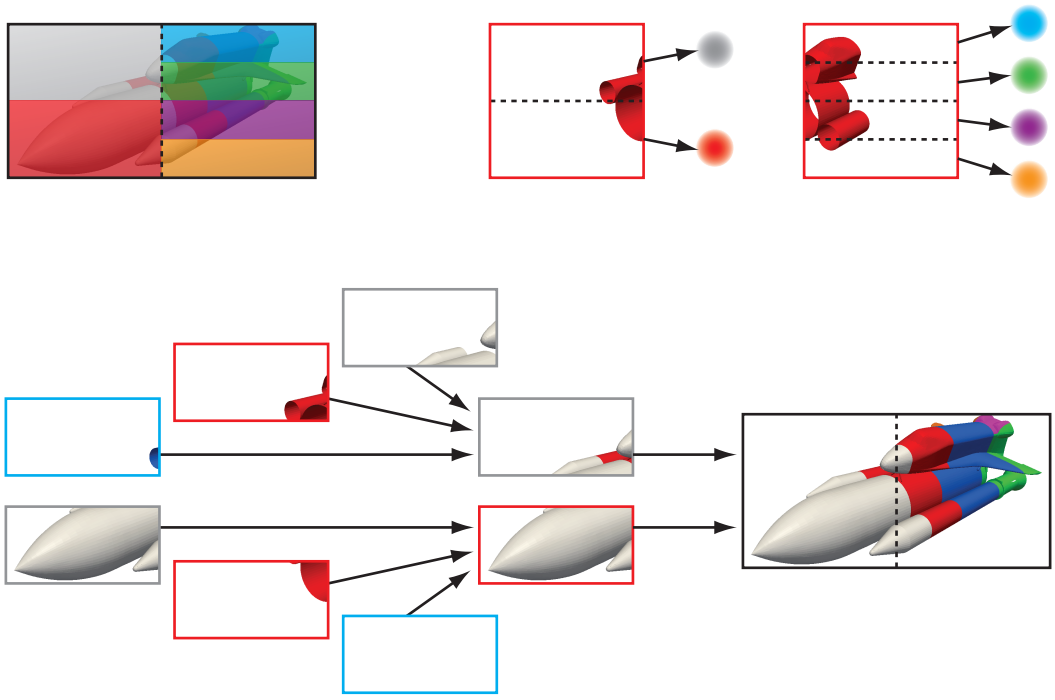

**Figure 5.5.** Compositing for split strategy. First tiles are split and assigned to processes (upper left). Then each process simultaneously sends its images to the responsible process (upper right) and receives all sub-images for its piece (bottom). The composited pieces are then collected and stitched together.

The reduce strategy supports ordered compositing. It does this by ensuring in the first phase that processes receive only images that are "near" the image they hold, that is, there is no other image in between the two images in the visibility ordering. The single image compositing algorithms of the second phase each support their own ordered compositing.

## Split Strategy

The **tile split and delegate strategy** (or simply the split strategy) is a simple algorithm that splits up tiles, assigns each piece to a tile, and then sends image fragments directly to the tiles for compositing. The split strategy makes efficient use of processing resources, but exhibits haphazard message passing which can cause issues on some high speed interconnects. The split strategy is selected by calling `icetStrategy` with the `ICET_STRATEGY_SPLIT` argument.

The split strategy first assigns processes to tiles similar to how they are assigned in the reduce strategy described previously. That is, the number of processes per tile is proportional to the num-

ber of non-empty images generated for it. Each tile is then split up evenly amongst all processes assigned to it. In the example in Figure 5.5, the upper left image shows that the left image is split between 2 processes and the right image is split amongst 4 processes.

On being assigned a section of tile, each process prepares to receive data from all the sending processes using asynchronous receives. Each process then renders its images, splits them up, and sends the sub-images to the corresponding process. When a process is ready and as it receives data, the incoming images are composited together. Once all of the incoming images are composited, the complete sub-image is sent to the display process to be stitched together.

The split strategy does not support ordered compositing. Using the split strategy in color blending mode will fail.

## Virtual Trees Strategy

The **virtual trees strategy** is based on the binary tree compositing algorithm, but performs multiple composites simultaneously to regain some of the load balance lost in the original algorithm. The virtual trees strategy has nice regular communications, but still suffers from some load imbalance, particularly when using fewer tiles and in later stages of the algorithm. The virtual trees strategy is selected by calling **icetStrategy** with the **ICET_STRATEGY_VTREE** argument.

The virtual trees strategy works by creating a "virtual" tree for each tile. Contained in each tree are processes that have rendered an image for that display tile. The algorithm proceeds much like the binary tree composition algorithm except that the processes float amongst the trees, helping with the compositing as they become available. Figure 5.6 shows an example of the virtual trees compositing. In particular, notice that the gray process takes part in the left tree in stage 1, then floats to take part in the right tree in stage 2, and then returns to take part in the left tree in stage 3.

When necessary, the process must keep track of multiple images belonging to different virtual trees. Two conserve memory, images are not rendered until they are needed. Also, a process can only hold two images at a time: one that it is sending and one that it is receiving. If a process is holding an image for one tile, it cannot receive an image for another tile until it sends away the image it is holding.

The virtual trees strategy does not support ordered compositing. Using the virtual trees strategy in color blending mode will fail.

## Serial Strategy

Despite its name, the **serial strategy** does not completely serialize the image compositing process. Rather, it serially addresses the tiles, but performs parallel compositing for each tile. The serial strategy is selected by calling **icetStrategy** with the **ICET_STRATEGY_SERIAL** argument.

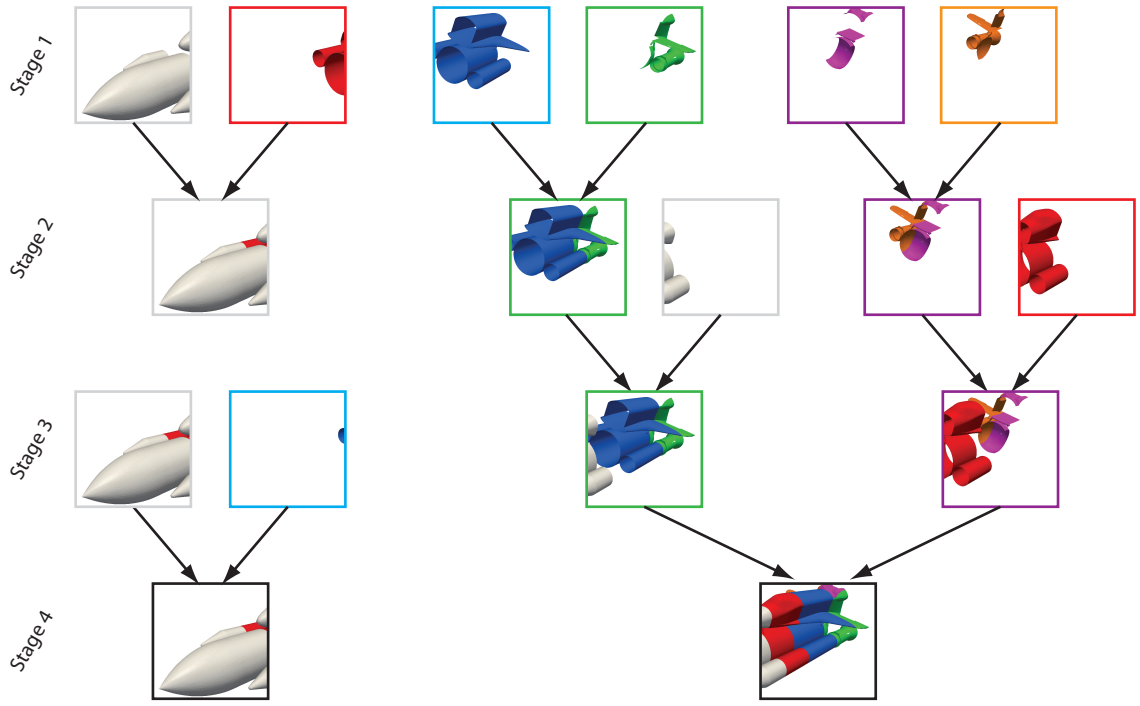


**Figure 5.6.** Composite network for virtual trees strategy. Arrows represent the passing of data from one state to the next. Processes receiving multiple images composite them together.

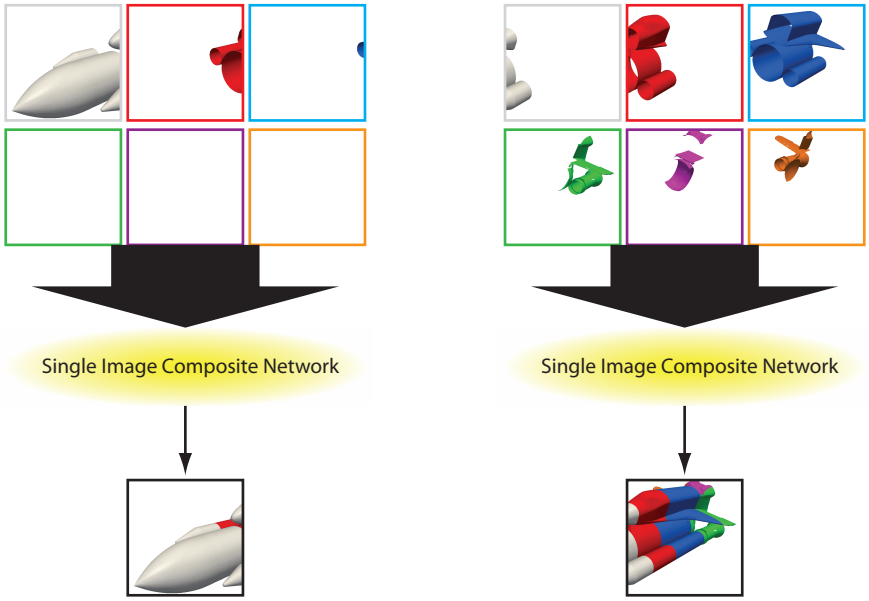


**Figure 5.7.** Composite network for serial compositing. One at a time, each tile is composited using a parallel single image composite network described in a previous section.

The serial strategy iterates over all of the tiles. For each tile, it composites all the images for that tile using one of the single image compositing algorithms described in that preceding section. As demonstrated in the example in Figure 5.7 images from all processes are composited for each tile regardless of whether some of them may be empty.

Since the single image compositing algorithms support ordered compositing, the serial strategy also supports ordered compositing.

The serial strategy is really only implemented as a baseline algorithm to compare other algorithms. In general, the reduce strategy does at least as well or outperforms the serial strategy. We have observed this even in single tile mode, possibly because the reduce strategy can throw away empty images.

## Direct Send Strategy

The **direct send strategy** is the simplest of all the strategies. Each process simply renders its images and sends them directly to the display process where the images get composited, as shown in Figure 5.8. The direct send strategy is selected by calling `icetStrategy` with the `ICET_STRATEGY_DIRECT` argument.

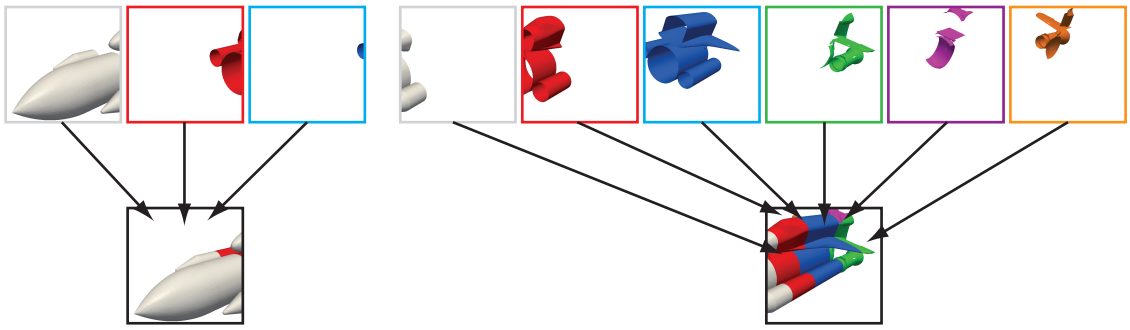

**Figure 5.8.** Composite network for direct send compositing. Arrows represent the passing of data from one process to another. Receiving process composite all incoming images together.

The direct send strategy is usually a poor performer. It was designed as a low watermark to compare to other compositing strategies. The direct send strategy does, however, support ordered compositing.

## Implementing New Strategies

The IceT API was written while its strategies were being developed. As such, the design yields for the relatively simplistic addition of new strategies. This section will provide the basic overview of how to add a new strategy. It is probably easiest to start by modifying your IceT source to insert your own strategy in the src/ice-t/strategies directory of the IceT source distribution. It should be possible to create an external library with a strategy in it, but there may be some complications with finding include files and getting the proper exports.

A strategy in IceT is created by simply defining an **`IceTStrategy`** object. The **`IceTStrategy`** type is generally meant to be an opaque type so that most users to not have to be concerned with the strategy innards. However, the IceT code base is currently quite stable and the **`IceTStrategy`** type rarely changes in any case.

The **`IceTStrategy`** type is defined in the `GL/ice-t.h` header file.

```
typedef GLuint *IceTImage;
typedef struct _IceTStrategy {
    const char *name;
    GLboolean supports_ordering;
    IceTImage (*compose)(void);
} IceTStrategy;
```

The `name` field in the **IceTStrategy** type is a short string identifying the strategy. (It is the string returned by **icetGetStrategyName** when the strategy is active.) The `supports_ordering` field is a Boolean value indicating whether the strategy supports ordered compositing. Finally, and most importantly, the `compose` field is a function that is called during an invocation of **icetDrawFrame** to perform the rendering and compositing. It takes no arguments (the composition should rely on the current IceT state), and returns a composited image. The **IceTImage** type is opaque to compositing algorithms, but several functions, discussed later in this chapter, are available to create and manipulate them.

All it takes to make IceT use a new strategy is to define and fill an object of type **IceTStrategy** and pass it to the **icetStrategy** function. Typically this is done as a static object somewhere within your library source. For example, the declaration for the direct send strategy looks like this.

```
static IceTImage directCompose(void);

IceTStrategy ICET_STRATEGY_DIRECT = { "Direct", ICET_TRUE, directCompose };
```

The object is also exposed in the `GL/ice-t.h` header file in a way that does not require linking directly to the strategy function.

```
ICET_STRATEGY_EXPORT extern IceTStrategy ICET_STRATEGY_DIRECT;
```

The **ICET_STRATEGY_DIRECT** name is intentionally formed to look like a C macro for an identifier, and it is intended to be used by the end user as such. (**ICET_STRATEGY_EXPORT** is a real C macro that performs library export magic.) Your strategy should define a **IceTStrategy** in a similar manner. The remainder of this section describes how to implement the compose function.

## INTERNAL STATE VARIABLES FOR COMPOSITING

As stated previously, a strategy's compose function does not take any arguments. Instead, it gets all relevant information from the IceT state. Many of the relevant state variables are described in the documentation for the **icetGet** functions (as well as elsewhere throughout this document). There are also several "hidden" state variables for internal use. The ones specifically useful for within a composite function are listed here (along with the variable type, number of entries, and a description). Note that these state variables generally should be read from, not written to.

**ICET_ALL_CONTAINED_TILES_MASKS** (boolean, **ICET_NUM_TILES** $\times$ **ICET_NUM_PROCESSES**) Contains an appended list of **ICET_CONTAINED_TILES_MASK** variables for all processes. Given process $p$ and tile $t$, the entry at $p + \textbf{ICET\_NUM\_TILES} \times t$ contains the flag describing whether process $p$ renders a non-blank image for tile $t$. This variable is the same on all processes.

**ICET_CONTAINED_TILES_LIST** (integer, **ICET_NUM_CONTAINED_TILES**) All the tiles into which the local geometry projects. In other words, this is the list of tiles which will not be empty after local rendering. The local processor should generate images for these tiles and participate in the composition of them.

**ICET_CONTAINED_TILES_MASK** (boolean, **ICET_NUM_TILES**) This is a list of boolean flags, one per tile. The flag is 1 if the local geometry projects onto the tile (that is, the local render will not be empty for that tile) and 0 otherwise. This gives the same information as **ICET_CONTAINED_TILES_LIST**, but in a different way that can be more convenient in some circumstances.

**ICET_CONTAINED_VIEWPORT** (integer, 4) Describes the region of the viewport that the geometry being rendered locally projects onto. The bounds of the data (given by **icetBoundingBox** or **icetBoundingVertices**) onto the tile display and determines the region of the tile display the data covers. The values in the four-tuple correspond to x, y, width, and height, respectively, of the projection in global pixel coordinates. This variable in conjunction with the **ICET_NEAR_DEPTH** and **ICET_FAR_DEPTH** give the full 3D projection of the local data in window space.

**ICET_FAR_DEPTH** (double, 1) The maximum depth value of the local geometry after projection. See **ICET_CONTAINED_VIEWPORTS** for more details.

**ICET_IS_DRAWING_FRAME** (boolean, 1) Set to true while in a call to **icetDrawFrame** and set to false otherwise. This should always be set to true while the compose function is being executed.

**ICET_NEAR_DEPTH** (double, 1) The minimum depth value of the local geometry after projection. See **ICET_CONTAINED_VIEWPORTS** for more details.

**ICET_NUM_CONTAINED_TILES** (integer, 1) The number of tiles into which the local geometry projects. This is the length of the **ICET_CONTAINED_TILES_LIST** variable.

**ICET_PROJECTION_MATRIX** (double, 16) The current projection matrix, read from OpenGL at the invocation of **icetDrawFrame**.

**ICET_TILE_CONTRIB_COUNTS** (integer, **ICET_NUM_TILES**) For each tile, provides the number of processes that will produce a non-empty image for that tile.

**ICET_TOTAL_IMAGE_COUNT** (integer, 1) The total number of non-empty images produced by all processes for all tiles. This variable is the sub of all entries in **ICET_TILE_CONTRIB_COUNTS**.

In addition to several internal state variables, IceT also has several internal functions for accessing them. The most important one for implementing a strategy is **icetUnsafeStateGet**, which is defined in the `state.h` header file.

```
void *icetUnsafeStateGet( GLenum pname );
```

The implementation for the **icetGet** functions is to copy the data into a memory buffer you provide, performing type conversion as necessary. **icetUnsafeStateGet** simply returns the internal pointer where the data is stored. This can be faster and more convenient (since you do not have to allocate your own memory), but is unsafe in two ways. First, if the state variable is changed, the pointer you receive can become invalid. Second, no type conversion is performed. You have to make sure you cast the pointer correctly yourself, and there is no real way to query the correct type. Since the state setting functions are hidden from the end user API, it is possible to manage these erroneous conditions.

## MEMORY MANAGEMENT

Compositing algorithms by their nature require buffers of memory of non-trivial size to hold images, among other data, that are not needed in between calls to the compositing. One approach is to simply use the standard C **malloc** and **free** functions. However, some implementations of **malloc**/**free** are not always efficient, and even the best implementations can have a tendency to fragment memory over time as large buffers are allocated and released.

To ensure efficient memory allocation, IceT provides its own memory management that is simple but effective for its compositing operations. IceT keeps around a pool of memory to be used by various components of the API. To use the **memory pool**, the code first clears the buffer and ensures that it is big enough. The code then reserves sections of the pool for various buffers. Since this pool changes size infrequently, allocating time or memory fragmentation is not an issue. The size of the allocated memory is also minimized since it is shared throughout all of IceT.

The **icetResizeBuffer** function is used to clear out the memory pool. The declaration for this function is located in the `context.h` header file.

```
void icetResizeBuffer( int size );
```

The **icetResizeBuffer** function ensures that the memory pool is at least *size* bytes large. It also resets all previously allocated memory (that is, freeing it back into the pool). This has two important consequences. First, you must know the amount of memory you need *a-priori*. You cannot resize the buffer once you have started allocating memory blocks. If you try to do so, the previously allocated blocks (potentially) will be destroyed.

Second, since this block of data is shared amongst all functions of IceT, you must be aware that other IceT code can potentially release your memory and allocate its own. You should feel free to use IceT's memory pool from within the compose function of your strategy and the image that it returns is best allocated from this buffer. Furthermore, the helper functions described in this section to implement your own strategy are also safe to call. Be aware, however, that in between calls to your composite function the memory you allocate will be lost and you will have to reallocate your buffers.

Once you have sized the memory pool, use **icetReserveBufferMem** to allocate a chunk of memory.

```
void *icetReserveBufferMem( int size );
```

**icetReserveBufferMem** returns a pointer to a buffer reserved to the given size. The buffer is aligned on 64-bit boundaries to help prevent illegal memory accesses.

The following code snippet (taken from the direct send strategy) demonstrates the use of **icetResizeBuffer** and **icetReserveBufferMem**. (The image size functions are described in the section on image functions.)

```
icetResizeBuffer(  2*icetSparseImageSize(max_pixels)
                 + icetFullImageSize(max_pixels)
                 + num_tiles*sizeof(GLint));
inImage     = icetReserveBufferMem(icetSparseImageSize(max_pixels));
outImage    = icetReserveBufferMem(icetSparseImageSize(max_pixels));
image       = icetReserveBufferMem(icetFullImageSize(max_pixels));
tile_image_dest = icetReserveBufferMem(num_tiles*sizeof(GLint));
```

## IMAGE MANIPULATION FUNCTIONS

You probably have noticed from the definition of the strategy structure that the compose function returns a variable of type **IceTImage**. There is another variable type used internally by strategies called **IceTSparseImage**. Both image types can hold color data or depth data or both. The **IceTImage** type stores pixels as raw data, simple 2D arrays that are compatible with OpenGL buffers. The **IceTSparseImage** stores images using active-pixel encoding, the run length encoding described in the Active-Pixel Encoding section of Chapter 4.

Both the **IceTImage** type and the **IceTSparseImage** type are opaque to compositing algorithms. Although you will create them by allocating a buffer and casting the pointer, you will not access the data directly. Instead, you will manipulate it with the functions described in this section. These functions are defined in the `image.h` header file.

### Creating Images

To create an image, you first need to know how big of a buffer you need. You can do this with the **icetFullImageSize** and **icetSparseImageSize** functions.

```
GLuint icetFullImageSize( GLuint pixels );

GLuint icetSparseImageSize( GLuint pixels );
```

The former of these functions return the size, in bytes, required for an **IceTImage** containing the number of *pixels* specified. The latter performs the same operation for an **IceTSparseImage**. A sparse image can vary in actual size depending on how well the data compresses so

**icetSparseImageSize** conservatively returns the maximum amount of bytes needed in any case.

To ensure memory is managed efficiently, your strategy will have to create all of the images it uses by allocating them with **icetResizeBuffer** and **icetReserveBufferMem** (discussed in the previous section with an example) and then casting the pointer to **IceTImage** or **IceTSparseImage** as appropriate.

Image structures are basically a block of memory with a small bit of header data. IceT functions that create images will fill that information for you. Occasionally you may need to explicitly fill the header information. This is done with **icetInitializeImage**.

```
void icetInitializeImage( IceTImage  image,
                          GLuint     pixel_count );
```

**icetInitializeImage** will initialize the *image* buffer to be a full containing *pixel_count* pixels and the type of pixel data specified by the **ICET_INPUT_BUFFERS** state parameter. There are only two common instances in which you will have to initialize an image yourself. The first is that you are filling the buffer one part at a time. The other is that you are creating a blank image, which frequently happens when a tile is empty. To clear out an image use **icetClearImage**.

```
void icetClearImage( IceTImage  image );
```

**icetClearImage** will set all of the pixel data in an image to the background. In practice, **icetClearImage** is coupled with a call to **icetInitializeImage** such as in the following.

```
image = icetReserveBufferMem(icetFullImageSize(max_pixels));
icetInitializeImage(image, max_pixels);
icetClearImage(image);
```

### Querying Images

Once you have an initialized image, whether initialized by you or some other IceT function, you can retrieve the number of pixels in it with **icetGetImagePixelCount**.

```
GLuint icetGetImagePixelCount( image );
```

Unlike most functions, **icetGetImagePixelCount** can take either a **IceTImage** or a **IceTSparseImage**.

If you need to access the actual data of a **IceTImage**, you can do so with **icetGetImageColorBuffer** and **icetGetImageDepthBuffer**.

```
GLubyte  *icetGetImageColorBuffer( IceTImage  image );
GLuint   *icetGetImageDepthBuffer( IceTImage  image );
```

For these functions to work, the image must be initialized and contain the respective color or depth buffer (of course). If this condition is not met, an error is raised and `NULL` is returned.

### Rendering Images

To get the image for a particular tile in the display, use either **icetGetTileImage** or **icetGetCompressedTileImage**.

```
void icetGetTileImage( GLint       tile,
                       IceTImage   buffer );

GLuint icetGetCompressedTileImage( GLint             tile,
                                   IceTSparseImage   buffer );
```

Both functions will invoke a rendering for that tile (performing the appropriate projection transformations) as necessary, read back the frame buffers and store the results in an image buffer you specify. The difference, of course, is that **icetGetTileImage** fills the buffer with raw data whereas **icetGetCompressedTileImage** will compress the image data with active-pixel encoding.

**icetGetTileImage** writes a pre-determined amount of data into *buffer*, which corresponds to the value returned by **icetFullImageSize**. The amount of data written to *buffer* by **icetGetCompressedTileImage** varies depending on how well the image compresses. The actual number of bytes written is returned by **icetGetCompressedTileImage**. In general, you should record this size as you will need it to transfer the data to another process. The amount of data written will never exceed the amount returned by **icetSparseImageSize**.

### Compressing Images

**icetCompressImage** converts a full **IceTImage** into to more compact **IceTSparseImage**.

```
GLuint icetCompressImage( const IceTImage   imageBuffer,
                          IceTSparseImage   compressedBuffer);
```

**icetCompressImage** returns the actual size of *compressedBuffer* in bytes.

Sometimes it is convenient to break up an image into pieces, and compress each piece. This is common when dividing up an image to be divided amongst some amount of processes. This can be most easily achieved by using the **icetCompressSubImage**.

```
GLuint icetCompressSubImage(
                        const IceTImage   imageBuffer,
                        GLuint            offset,
                        GLuint            pixels,
                        IceTSparseImage   compressedBuffer);
```

**icetCompressSubImage** compresses a region of contiguous pixels. The block of pixels starts *offset* pixels past the beginning of the image and is *pixels* long. *icetCompressImage* is equivalent to calling *icetCompressSubImage* with *offset* set to 0 and *pixels* set to the result of **icetGetImagePixelCount**.

A sparse image can be returned to its uncompressed form with **icetDecompressImage**.

```
GLuint icetDecompressImage(
                        const IceTSparseImage  compressedBuffer,
                        IceTImage              imageBuffer  );
```

**icetDecompressImage** returns the number of pixels in the resulting image, which is the same number that you will get if you call **icetGetImagePixelCount** on the resulting image.

## COMMUNICATIONS

The first thing to know about communications in IceT is to understand that it is up to the strategy to count how many bytes are being transmitted in your compose function and store this in the **ICET_BYTES_SENT** state variable. To make this easier, `common.h` (found in the strategies directory) provides **icetAddSentBytes**.

```
void icetAddSentBytes( GLint  num_sending  );
```

**icetAddSentBytes** simply adds *num_sending* to the value in state variable **ICET_BYTES_SENT**. A call to **icetAddSentBytes** should be coupled with every communication call that sends data.

IceT provides an abstract communication layer, which is described in detail in Chapter 6. A handle to a communicator is stored in the current context. To make using the communicator easier, a set of convenience functions described next is available in the `context.h` include file. All of these functions are based off of those found in the MPI standard. For documentation, see that for the corresponding MPI function. Note that each function is missing an argument specifying the communicator. These functions just grab the current context's communicator.

```
struct IceTCommunicatorStruct *ICET_COMM_DUPLICATE(void);

void ICET_COMM_DESTROY(void);
```

```
void ICET_COMM_SEND( const void *  buf,
                     int           count,
                     GLenum        datatype,
                     int           dest,
                     int           tag  );

void ICET_COMM_RECV( void *  buf,
                     int     count,
                     GLenum  datatype,
                     int     src,
                     int     tag  );

void ICET_COMM_SENDRECV( const void *  sendbuf,
                         int           sendcount,
                         GLenum        sendtype,
                         int           dest,
                         int           sendtag,
                         void *        recvbuf,
                         int           recvcount,
                         GLenum        recvtype,
                         int           src,
                         int           recvtag  );

void ICET_COMM_ALLGATHER( const void *  sendbuf,
                          int           sendcount,
                          GLenum        type,
                          void *        recvbuf  );

IceTCommRequest ICET_COMM_ISEND( const void *  buf,
                                 int           count,
                                 GLenum        datatype,
                                 int           dest,
                                 int           tag  );

IceTCommRequest ICET_COMM_IRECV( void *  buf,
                                 int     count,
                                 GLenum  datatype,
                                 int     src,
                                 int     tag  );

void ICET_COMM_WAIT(  IceTCommRequest *  request  );

void ICET_COMM_WAITANY(
                     int                 count,
                     IceTCommRequest *   array_of_requests  );
```

```
int ICET_COMM_SIZE(void);

int ICET_COMM_RANK(void);
```

In each of these functions, the type parameter is set to one of the following: **ICET_BOOLEAN**, **ICET_BYTE**, **ICET_SHORT**, **ICET_INT**, **ICET_FLOAT**, or **ICET_DOUBLE**

### Transferring Images

Although the **IceTImage** and **IceTSparseImage** types are opaque, they can be transferred as simple byte buffers. To do so, you need only the size of the buffer. The following sends an image stored in the variable `image` of type **IceTImage**.

```
size = icetFullImageSize(icetGetImagePixelCount(image));
icetAddSentBytes(size);
ICET_COMM_SEND(image, size, ICET_BYTE, dest, tag);
```

And the following is the paired receive for the image. Note that the number of pixels in `pixel_count` need to be as large or larger then the actual number of pixels sent, but it otherwise does not have to match exactly. And, of course, `image` must be allocated (generally with **icetResizeBuffer** and **icetReserveBufferMem**) with the appropriate amount of memory.

```
size = icetFullImageSize(pixels);
ICET_COMM_RECV(image, size, ICET_BYTE, src, tag);
```

Most of the time, you will actually send compressed image data. Compressed images are sent in the same manner as full image. The only difference is to make sure you give the communication function the actual size of the image.

```
size = icetCompressImage(image, compressed_image);
ICET_COMM_SEND(compressed_image, size, ICET_BYTE, dest, tag);
```

And the following is the paired receive for the sparse image. Note that we do not need to know the actual number of bytes received. Rather, we just need to know the maximum size of the image and have a buffer that large.

```
size = icetSparseImageSize(pixels);
ICET_COMM_RECV(compressed_image, size, ICET_BYTE, src tag);
```

### Helper Communication Functions

common.h (found in the strategies directory) contains some helper functions that implement common communication patterns. They may be helpful in implementing your strategy.

```
void icetRenderTransferFullImages(
                          IceTImage         imageBuffer,
                          IceTSparseImage   inImage,
                          IceTSparseImage   outImage,
                          GLint             num_receiving,
                          GLint *           tile_image_dest  );
```

**icetRenderTransferFullImages** renders all the tiles that are specified in the **ICET_CONTAINED_TILES** state array and sends them to the processors with ranks specified in *tile_image_dest*. This method is guaranteed not to deadlock. It only uses memory given with the buffer arguments, and will make its best efforts to get the graphics and network hardware to run in parallel.

*imageBuffer* is a buffer big enough to hold color and/or depth values that is **ICET_MAX_PIXELS** big. The size can be determined with the **icetFullImageSize** function in image.h. *inImage* and *outImage* are two buffers big enough to hold sparse color and depth information for an image that is **ICET_MAX_PIXELS** big. The size can be determined with the **icetSparseImageSize** macro in image.h. *num_receiving* is the number of images this processor is receiving, and *tile_image_dest* is an array where if tile $t$ is in **ICET_CONTAINED_TILES**, then the rendered image for tile $t$ is sent to *tile_image_dest*$[t]$.

There is also a more general form for transferring images or other large data blocks.

```
typedef void *(*IceTGenerateData)(GLint id, GLint dest, GLint *size);

typedef void *(*IceTHandleData)(void *, GLint src);

void icetSendRecvLargeMessages(
                          GLint               numMessagesSend,
                          GLint *             messageDestinations,
                          GLint               messagesInOrder,
                          IceTGenerateData    generateDataFunc,
                          IceTHandleData      handleDataFunc,
                          void *              incomingBuffer,
                          GLint               bufferSize);
```

**icetSendRecvLargeMessages** is similar to **icetRenderTransferFullImages** except that it works with generic data, data generators, and data handlers. It takes a count of a number of messages to be sent and an array of ranks to send to. Two callbacks are required. One generates the data (so large data may be generated JIT to save memory) and the other handles incoming data. The generate callback is run right before the data it returns is sent to a particular destination. This callback will not be called again until the memory it returned is no longer in use,

so the memory may be reused. As large messages come in, the handle callback is called. As an optimization, if a process sends to itself, then that will be the first message created. This gives the callback an opportunity to build its local data while waiting for incoming data. The handle callback returns a pointer to a buffer to be used for the next large message receive. It should be common for this message buffer to be reused too.

*numMessagesSending* is a count of the number of large messages this processor is sending out. *messageDestinations* is an array of size *numMessagesSending* that contains the ranks of message destinations. *generateDataFunc* is a callback function that generates messages. The function is given the index in *messageDestinations* and the rank of the destination as arguments. The data of the message and the size of the message (in bytes) are returned. The *generateDataFunc* will not be called again until the returned data is no longer in use. Thus the data may be reused. *handleDataFunc* is a callback function that processes messages. The function is given the data buffer and the rank of the process that sent it. The function is expected to return a buffer to use for the next message receive. If the callback is finished with the buffer it was given, it is perfectly acceptable to return it again for reuse. *incomingBuffer* is a buffer to use for the first incoming message. *bufferSize* is the maximum size of a message.

## INTERNAL FUNCTIONS FOR COMPOSITING

In the strategies directory, the `common.h` header has prototypes for the single image compositing algorithms described in the Single Image Compositing section of this chapter.

### Parallel Compositing

```
void icetTreeCompose(  GLint *            compose_group,
                       GLint              group_size,
                       GLint              image_dest,
                       IceTImage          imageBuffer,
                       IceTSparseImage    compressedImageBuffer  );
```

**icetTreeCompose** performs a binary tree composition of images amongst a subset of processes in the current communicator of the context. Rather than perform the composition on all the processes in the communicator, it performs them on a subset with arbitrary ordering. (Note that ordering matters when doing alpha blending as opposed to the z-buffer operation.) *compose_group* is the mapping of processes from the communicator ranks to the "group" ranks. The size of the groups (and the length of the *compose_group* array) is specified by *group_size*. The compose image ends up in the processor with rank *compose_group*[*image_dest*]. *imageBuffer* should contain the partial input image to be composited. (Of course, each process should have its own partial image. All processes should provide images of identical dimensions.) On the process with the rank *compose_group*[*image_dest*], the final image will be stored in this buffer. *compressedImageBuffer* is a buffer that is used internally by **icetTreeCompose** for compressing, sending, and receiving images. It must be large enough to hold an image as large

as the input, but its contents are ignored on the function invocation and the contents are garbage on return.

The following is a very simple example of compositing the image on tile 0 and providing the result on the process with rank 0. If ordered compositing is enabled, then the order is respected. This is similar to the serial strategy except that only the first tile is composited.

```
IceTImage treeComposeTile0(void)
{
  GLint max_pixels;
  GLint rank;
  GLint num_proc;
  GLint *display_node;
  GLint image_dest;
  GLboolean ordered_composite;
  IceTImage image;
  IceTSparseImage scratchImage;
  GLint *compose_group;
  int i;

  icetGetIntegerv(ICET_NUM_TILES, &num_tiles);
  icetGetIntegerv(ICET_TILE_MAX_PIXELS, &max_pixels);
  icetGetIntegerv(ICET_RANK, &rank);
  icetGetIntegerv(ICET_NUM_PROCESSES, &num_proc);
  display_nodes = icetUnsafeStateGet(ICET_DISPLAY_NODES);
  ordered_composite = icetIsEnabled(ICET_ORDERED_COMPOSITE);

  icetResizeBuffer(  icetFullImageSize(max_pixels)
                   + icetSparseImageSize(max_pixels)
                   + sizeof(int)*num_proc);
  image         = icetReserveBufferMem(icetFullImageSize(max_pixels));
  scratchImage  = icetReserveBufferMem(icetSparseImageSize(max_pixels));
  compose_group = icetReserveBufferMem(sizeof(GLint)*num_proc);

  if (ordered_composite) {
    icetGetIntegerv(ICET_COMPOSITE_ORDER, compose_group);
    for (image_dest = 0; compose_group[image_dest] != display_nodes[i];
         image_dest++);
  } else {
    for (i = 0; i < num_proc; i++) {
      compose_group[i] = i;
    }
    image_dest = display_nodes[0];
  }

  icetGetTileImage(0, image);
  icetTreeCompose(compose_group, num_proc, image_dest, image, scratchImage);
```

```
  return image;
}
```

A much more scalable image compositing algorithm is binary swap. Usually you will use the binary-swap algorithm instead of tree compose. The only exception is that binary-swap has a bit more overhead than tree compose, so for small amounts of processes it may be moderately faster to run tree compose.

```
void icetBswapCompose( GLint *            compose_group,
                       GLint              group_size,
                       GLint              image_dest,
                       IceTImage          imageBuffer,
                       IceTSparseImage    scratchImage1,
                       IceTSparseImage    scratchImage2  );
```

**icetBswapCompose** behaves very much like **icetTreeCompose** except that it uses a different (and much more scalable) algorithm. The arguments of the two functions are very similar. (The only difference is that **icetBswapCompose** requires two **IceTSparseImage** buffers whereas **icetTreeCompose** requires only one.)

Rather than perform the composition on all the processes in the communicator, **icetBswapCompose** performs them on a subset with arbitrary ordering. (Note that ordering matters when doing alpha blending as opposed to the z-buffer operation.) *compose_group* is the mapping of processes from the communicator ranks to the "group" ranks. The size of the groups (and the length of the *compose_group* array) is specified by *group_size*. The compose image ends up in the processor with rank *compose_group*[*image_dest*]. *imageBuffer* should contain the partial input image to be composited. (Of course, each process should have its own partial image. All processes should provide images of identical dimensions.) On the process with the rank *compose_group*[*image_dest*], the final image will be stored in this buffer. *scratchImage1* and *scratchImage2* are buffers that are used internally by **icetBswapCompose** for compressing, sending, and receiving images. It must be large enough to hold an image as large as the input, but its contents are ignored on the function invocation and the contents are garbage on return.

The following is a very simple example of compositing the image on tile 0 and providing the result on the process with rank 0. It is identical to the previous example code except that it uses binary swap and is equally similar to the serial strategy. If ordered compositing is enabled, then the order is respected.

```
IceTImage bswapComposeTile0(void)
{
  GLint max_pixels;
  GLint rank;
  GLint num_proc;
```

```
  GLint *display_node;
  GLint image_dest;
  GLboolean ordered_composite;
  IceTImage image;
  IceTSparseImage scratchImage1, scratchImage2;
  GLint *compose_group;
  int i;

  icetGetIntegerv(ICET_NUM_TILES, &num_tiles);
  icetGetIntegerv(ICET_TILE_MAX_PIXELS, &max_pixels);
  icetGetIntegerv(ICET_RANK, &rank);
  icetGetIntegerv(ICET_NUM_PROCESSES, &num_proc);
  display_nodes = icetUnsafeStateGet(ICET_DISPLAY_NODES);
  ordered_composite = icetIsEnabled(ICET_ORDERED_COMPOSITE);

  icetResizeBuffer(  icetFullImageSize(max_pixels)
                   + icetSparseImageSize(max_pixels)*2
                   + sizeof(int)*num_proc);
  image         = icetReserveBufferMem(icetFullImageSize(max_pixels));
  scratchImage1 = icetReserveBufferMem(icetSparseImageSize(max_pixels));
  scratchImage2 = icetReserveBufferMem(icetSparseImageSize(max_pixels));
  compose_group = icetReserveBufferMem(sizeof(GLint)*num_proc);

  if (ordered_composite) {
    icetGetIntegerv(ICET_COMPOSITE_ORDER, compose_group);
    for (image_dest = 0; compose_group[image_dest] != display_nodes[i];
         image_dest++);
  } else {
    for (i = 0; i < num_proc; i++) {
      compose_group[i] = i;
    }
    image_dest = display_nodes[0];
  }

  icetGetTileImage(0, image);
  icetBswapCompose(compose_group, num_proc, image_dest, image,
                   scratchImage1, scrachImage2);

  return image;
}
```

### Local Compositing

When developing a multi-tile compositing algorithm (or any parallel compositing algorithm for that matter), it is often cannot be broken into full composites of single images. Instead, you must break the problem down further into image transfers and image combinations. Image transfers

have already been covered previously in this section. The IceT library contains multiple methods to locally composite two images together.

```
void icetComposite( IceTImage       destBuffer,
                    const IceTImage srcBuffer,
                    int             srcOnTop  );
```

**icetComposite** takes the images stored in *destBuffer* and *srcBuffer*, composites them together, and stores the result in *destBuffer*. The compositing operation is automatically determined by the current state. (See Chapter 4 for information on how the compositing operation is determined.) If the compositing operation is order dependent, then the Boolean argument *srcOnTop* determines whether *srcBuffer* or *destBuffer* is on top.

If one of your images is compressed (stored in a **IceTSparseImage**, it is faster to perform the compositing operation on the compressed image rather than decompressing first. In fact, it is faster to composite a compressed image than two full image because the active-pixel encoding allows the composite algorithm to skip over groups of background pixels. This gives you the double win of faster image transfer and faster compositing.

```
void icetCompressedComposite(
                          IceTImage             destBuffer,
                          const IceTSparseImage srcBuffer,
                          int                   srcOnTop  );
```

**icetCompressedComposite** behaves just like **icetComposite** except that *srcBuffer* is a compressed image rather than a full image. The images in *destBuffer* and *srcBuffer* are composited together, and the results are stored in *destBuffer*.

Many parallel compositing algorithms break images into pieces, distribute amongst processes, and composite the pieces. To facilitate the compositing image pieces, IceT provides **icetCompressedSubComposite**.

```
void icetCompressedSubComposite(
                          IceTImage             destBuffer,
                          GLuint                offset,
                          GLuint                pixels,
                          const IceTSparseImage srcBuffer,
                          int                   srcOnTop);
```

The *destBuffer*, *srcBuffer* and *srcOnTop* arguments are the same as those in **icetCompressedComposite**. The *offset* and *pixels* arguments specify a region of contiguous pixels in *destBuffer* to perform the compositing in.

# Chapter 6

# Communicators

IceT implements an abstract communication layer. As we will see later in this chapter, this communication layer is a message passing interface based heavily on MPI.[1] As an end user to IceT, you need to know almost nothing about this communication layer. You need only to get a reference to an **IceTCommunicator** object. This object is opaque. You only need to get one, pass it to the **icetCreateContext**, and then delete it. **icetCreateContext** will duplicate the communicator, so you need not worry about when you delete the context you created.

Most of the time you will use the built-in MPI implementation of the communicator, which is discussed in the first section. If necessary, you can write your own communicator, which is discussed in the following section.

## MPI Communicators

Using the MPI implementation of a communicator, you simply include `GL/ice-t_mpi.h` in your source and link icet_mpi into your own library or executable. The only function you need to use is **icetCreateMPICommunicator**.

```
IceTCommunicator icetCreateMPICommunicator(
                                        MPI_Comm mpi_comm );
```

Quite simply, **icetCreateMPICommunicator** converts an **MPI_Comm**, an MPI communicator, into an **IceTCommunicator**, an IceT communicator. **icetCreateMPICommunicator** duplicates the MPI communicator. Thus, you can delete the *mpi_comm* communicator as soon as **icetCreateMPICommunicator** as soon as it exits. Furthermore, the returned **IceTCommunicator** will internally manage the MPI communicator it created.

Once created, the **IceTCommunicator** may be deleted with **icetDestroyMPICommunicator**.

```
void icetDestroyMPICommunicator( IceTCommunicator comm );
```

[1] In fact, the original implementation of IceT used MPI directly. The abstract layer was inserted later as a more-or-less cut-and-paste operation.

**icetDestroyMPICommunicator** will release all the resources used by *comm*. This includes the internal MPI communicator, which you do not have direct access to. *comm* will be invalid once you call **icetDestroyMPICommunicator**. However, you do not have to worry about any IceT context you have passed it to since they will have duplicated the communicator.

Using the MPI communicator is easy. First, you include the `GL/ice-t_mpi.h` header.

```
#include <GL/ice-t.h>
#include <GL/ice-t_mpi.h>
```

When you are ready to create an IceT context (usually during the initialization of your program), create the MPI-based communicator, use it to initialize the context, and then destroy the communicator.

```
icetComm = icetCreateMPICommunicator(MPI_COMM_WORLD);
icetContext = icetCreateContext(icetComm);
icetDestroyMPICommunicator(icetComm);
```

Once you have a context, you can use IceT as explained throughout this document. When you are ready, destroy the context as you normally would.

```
icetDestroyContext(icetContext);
```

Finally, do not forget to use the icet_mpi library when linking your executable or library.

A more detailed example of using the MPI communicator is in the Chapter 2 tutorial.

# User Defined Communicators

Occasionally, it may be necessary to provide your own version of a parallel communicator. This may be because you are using a communication library other than MPI. It may also be because you wish to augment the behavior of MPI when it is used by IceT. To provide your own communicator, you need only to create an **IceTCommunicator** object. In previous sections we have discussed **IceTCommunicator** as an opaque type, and unless you are implementing your own you should treat it as such. If you are implementing a **IceTCommunicator**, you will see that it is simply a pointer to a structure containing references to several communication functions.

```
struct IceTCommunicatorStruct {
    struct IceTCommunicatorStruct *
         (*Duplicate)(struct IceTCommunicatorStruct *self);
```

```
    void (*Destroy)(struct IceTCommunicatorStruct *self);
    void (*Send)(struct IceTCommunicatorStruct *self,
                 const void *buf, int count, GLenum datatype, int dest,
                 int tag);
    void (*Recv)(struct IceTCommunicatorStruct *self,
                 void *buf, int count, GLenum datatype, int src, int tag);

    void (*Sendrecv)(struct IceTCommunicatorStruct *self,
                     const void *sendbuf, int sendcount, GLenum sendtype,
                     int dest, int sendtag,
                     void *recvbuf, int recvcount, GLenum recvtype,
                     int src, int recvtag);
    void (*Allgather)(struct IceTCommunicatorStruct *self,
                      const void *sendbuf, int sendcount, int type,
                      void *recvbuf);

    IceTCommRequest (*Isend)(struct IceTCommunicatorStruct *self,
                             const void *buf, int count, GLenum datatype,
                             int dest, int tag);
    IceTCommRequest (*Irecv)(struct IceTCommunicatorStruct *self,
                             void *buf, int count, GLenum datatype,
                             int src, int tag);

    void (*Wait)(struct IceTCommunicatorStruct *self, IceTCommRequest *request);
    int  (*Waitany)(struct IceTCommunicatorStruct *self,
                    int count, IceTCommRequest *array_of_requests);

    int  (*Comm_size)(struct IceTCommunicatorStruct *self);
    int  (*Comm_rank)(struct IceTCommunicatorStruct *self);
    void *data;
};

typedef struct IceTCommunicatorStruct *IceTCommunicator;
```

To create a custom **`IceTCommunicator`** simply allocate the structure and fill in the function pointers. An implementation for a function that creates an IceT communicator might look like the following. In this example, the `my*` functions are implementations of the communication functions.

```
IceTCommunicator myCreateCommunicator(myCommType myComm)
{
  IceTCommunicator comm = malloc(sizeof(struct IceTCommunicatorStruct));

  comm->Duplicate = myDuplicate;
  comm->Destroy = myDestroy;
  comm->Send = mySend;
  /* And so on... */
```

```
  comm->data = malloc(sizeof(myComm))
  /* Making a duplicate here would be better. */
  memcpy(comm->data, myComm, sizeof(myComm));

  return comm;
}
```

The paired destruction function should probably just call the Destroy function of the communicator (or vice versa) to ensure that destroy works either way.

```
void myDestroyCommunicator(IceTCommunicator comm)
{
  comm->Destroy(comm);
}


static void myDestroy(IceTCommunicator self)
{
  myCommType *myComm = (myCommType *)self->data;
  /* Release resources of myComm. */
  free(myComm);
  free(self);
}
```

For a more concrete example of implementing an IceT communicator, see the IceT code for the MPI communicator.

# Chapter 7

# Future Work

The majority of the development for IceT was finished by 2004. Since then, IceT has proven to be a stable and versatile library that is currently being used in several production applications.

The following is a list of potential changes to IceT. As of this writing, none of these are currently under development. Rather, these are identified shortcomings of various degrees in IceT. These features will be handled on an as needed basis, assuming the need should arise.

**Update scalability tests.** The scalability tests for IceT were run during its major development, which was several years ago. At the time, 64 nodes was considered a pretty enormous big visualization cluster. Nowadays, IceT is sometimes used to run visualization on supercomputers containing thousands of processes. Verifying the scalability of the system occasionally is always a good idea. Also, the reduce strategy makes assumptions about the relative performance of the binary tree and binary swap composition algorithms. The performance point might have changed. Also, given that binary swap is reducing the image with each step, it might be worthwhile to switch to binary tree in the middle of the binary swap algorithm.

**Render aborts.** In interactive applications, it is often convenient to be able to abort a render that takes some time to finish. Aborting a render in the middle of a composite is tricky, because you need to make sure that everyone is aware of the abort and that all communication is correctly canceled. This could be partially implemented in IceT's communication layer, but all the strategies still have to be ready to quit once a communication is canceled due to an abort (or at the very least ignore it without crashing).

**Decouple from OpenGL.** In retrospect, tying the IceT library so closely to OpenGL was a mistake. Although most graphical applications, even today, use OpenGL for their rendering, it unnecessarily precludes the use of any other graphics library (such as DirectX, for example) or any application that does not use a graphics API to create images. The direct interfacing with OpenGL can even complicate the integration with applications that also use OpenGL. Although there should always be a layer to simplify the interface with OpenGL, ideally you should be able to use a compositing library like IceT independently of OpenGL.

**Allow higher precision color buffers.** The internal representation of images in IceT uses 8-bit colors. This was an intentional design so that RGBA colors can be treated as single 32-bit integers and greatly speed up compositing operations. Even today, this is fine representation for colors. Few display devices can provide more color resolution. However, when blending

colors stored as 8-bit values, quantization errors can occur. Most of the time this is not an issue with IceT, but it can create seams in images with very low opacity.

**Automatically count network communication.** The way IceT builds the communication metric stored in `ICET_BYTES_SENT` is to require each strategy to count how many bytes they send. This is a rather fragile implementation. A better approach would be to have the communications layer count bytes.

**Source file names.** The names of the internal IceT source files are a bit too generic. Names like image.c and draw.c are not clearly part of IceT and might conflict with other source file names in projects where the libraries are used. Although this will not cause any namespace collisions in linking, it can be irritating while debugging if the file names are not easily resolved.

# Chapter 8

# Man Pages

In this chapter you will find a man page for each of the functions available in the IceT API.

## NAME

**icetAddTile** – add a tile to the logical display.

## SYNOPSIS

```
#include <GL/ice-t.h>

    int icetAddTile(  GLint    x,
                      GLint    y,
                      GLsizei  width,
                      GLsizei  height,
                      int      display_rank  );
```

## DESCRIPTION

Adds a tile to the tiled display. Every process, whether actually displaying a tile or not, must declare the tiles in the display and which processes drive them with **icetResetTiles** and **icetAddTile**. Thus, each process calls **icetAddTile** once for each tile in the display, and all processes must declare them in the same order.

The parameters *x*, *y*, *width*, and *height* define the tiles viewport in the logical global display much in the same way **glViewport** declares a region in a physical display. IceT places no limits on the extents of the logical global display. That is, there are no limits on the values of *x* and *y*. They can extend as far as they want in both the positive and negative directions.

IceT will project its images onto the region of the logical global display that just covers all of the tiles. Therefore, shifting all the tiles in the logical global display by the same amount will have no real overall effect.

The *display_rank* parameter identifies the rank of the process that will be displaying the given tile. It is assumed that the output of the rendering window of the given process is projected onto the space in a tiled display given by *x*, *y*, *width*, and *height*. Each tile must have a valid rank (between 0 and **ICET_NUM_PROCESSES** − 1). Furthermore, no process may be displaying more than one tile.

## RETURN VALUE

Returns the index of the tile created.

## ERRORS

**ICET_INVALID_VALUE** Raised if *display_rank* is not a valid process rank or *display_rank* is already assigned to another tile. If this error is raised, nothing is done and -1 is returned.

## WARNINGS

None.

## BUGS

**icetAddTile** will let you add tiles of different sizes, but the use of different sized tiles is not yet supported. The user should try to make sure that all tiles are of the same size.

All processes must specify the same tiles in the same order. IceT will assume this even though it is not explicitly detected or enforced.

## COPYRIGHT



## SEE ALSO

**icetResetTiles**

## NAME

**icetBoundingBoxd**, **icetBoundingBoxf** – set bounds of geometry

## SYNOPSIS

```
#include <GL/ice-t.h>

    void icetBoundingBoxd ( GLdouble  x_min,
                            GLdouble  x_max,
                            GLdouble  y_min,
                            GLdouble  y_max,
                            GLdouble  z_min,
                            GLdouble  z_max  );

    void icetBoundingBoxf ( GLfloat  x_min,
                            GLfloat  x_max,
                            GLfloat  y_min,
                            GLfloat  y_max,
                            GLfloat  z_min,
                            GLfloat  z_max  );
```

## DESCRIPTION

Establishes the bounds of the geometry as contained in an axis-aligned box with the given extents.

**icetBoundingBoxd** and **icetBoundingBoxf** are really just convience functions. They create an array of the 8 corner vertices and set the bounding vertices appropriately. See **icetBoundingVertices** for more information.

## ERRORS

None.

## WARNINGS

None.

## BUGS

None known.

## COPYRIGHT



## SEE ALSO

**`icetBoundingVertices`**

## NAME

**icetBoundingVertices** – set bounds of geometry.

## SYNOPSIS

```
#include <GL/ice-t.h>

    void icetBoundingVertices( GLint           size,
                               GLenum          type,
                               GLsizei         stride,
                               GLsizei         count,
                               const GLvoid *  pointer  );
```

## DESCRIPTION

**icetBoundingVertices** is used to tell IceT what the bounds of the geometry drawn by the callback registered with **icetDrawFunc** are. The bounds are assumed to be the convex hull of the vertices given. The user should take care to make sure that the drawn geometry actually does fit within the convex hull, or the data may be culled in unexpected ways. IceT runs most efficiently when the bounds given are tight (match the actual volume of the data well) and when the number of vertices given is minimal.

The *size* parameter specifies the number of coordinates given for each vertex. Coordinates are given in X-Y-Z-W order. Any Y or Z coordinate not given (because *size* is less than 3) is assumed to be 0.0, and any W coordinate not given (because *size* is less than 4) is assumed to be 1.0.

The *type* parameter specifies in what data type the coordinates are given. Valid *type*s are **ICET_SHORT**, **ICET_INT**, **ICET_FLOAT**, and **ICET_DOUBLE**, which correspond to types `GLshort`, `GLint`, `GLfloat`, and `GLdouble`, respectively.

The *stride* parameter specifies the offset between consecutive vertices in bytes. If *stride* is 0, the array is assumed to be tightly packed.

The *count* parameter specifies the number of vertices to set.

The *pointer* parameter is an array of vertices with the first vertex starting at the first byte.

If data replication is being used, each process in a data replication group should register the same bounding vertices that encompass the entire geometry. By default there is no data replication, so unless you call **icetDataReplicationGroup**, all process can have their own bounds.

## ERRORS

| | |
|---|---|
| **ICET_INVALID_VALUE** | Raised if *type* is not one of **ICET_SHORT**, **ICET_INT**, **ICET_FLOAT**, or **ICET_DOUBLE**. |

## WARNINGS

None.

## BUGS

None known.

## COPYRIGHT



## SEE ALSO

**icetDrawFunc**, **icetBoundingBox**, **icetDataReplicationGroup**

## NAME

**icetCompositeOrder** – specify the order in which images are composited

## SYNOPSIS

```
#include <GL/ice-t.h>

   void icetCompositeOrder( const GLint * process_ranks );
```

## DESCRIPTION

If **ICET_ORDERED_COMPOSITE** is enabled and the current strategy supports ordered composition, then the order which images are composited are specified with **icetCompositeOrder**. If compositing is done with z-buffer comparisons (i.e. the depth buffer is selected as an input with **icetInputOutputBuffers**), then the ordering does not matter, and **ICET_ORDERED_COMPOSITE** should probably be disabled. However, if compositing is done with color blending (i.e. the depth buffer is *not* selected as in input with **icetInputOutputBuffers**), then the order in which the images are composed can drastically change the output.

For ordered image compositing to work, the geometric objects rendered by processes must be arranged such that if the geometry of one process is "in front" of the geometry of another process for any camera ray, that ordering holds for all camera rays. It is the application's responsibility to ensure that such an ordering exists and to find that ordering. The easiest way to do this is to ensure that the geometry of each process falls cleanly into regions of an octree, k-d tree, or similar structure.

Once the geometry order is determined for a particular rendering viewpoint, it is given to IceT in the form of an array of ranks. The parameter *process_ranks* should have exactly **ICET_NUM_PROCESSES** entries, each with a unique, valid process rank. The first process should have the geometry that is "in front" of all others, the next directly behind that, and so on. It should be noted that the application may actually impose only a partial order on the geometry, but that can easily be converted to the linear ordering requird by IceT.

When ordering is on, it is accepted that **icetCompositeOrder** will be called in between every frame since the order of the geometry may change with the viewpoint.

If data replication is in effect (see **icetDataReplicationGroup**), all processes are still expected to be listed in *process_ranks*. Correct ordering can be achieved by ensuring that all processes in each group are listed in contiguous entries in *process_ranks*.

## ERRORS

| | |
|---|---|
| **ICET_INVALID_VALUE** | Not every entry in the parameter *process_ranks* was a unique, valid process rank. |

## WARNINGS

None.

## BUGS

If an **ICET_INVALID_VALUE** error is raised, internal arrays pertaining to the ordering of images may not be restored properly. If such an error is raised, the function should be re-invoked with a valid ordering before preceding. Unpredictable results may occur otherwise.

## COPYRIGHT



## SEE ALSO

**icetInputOutputBuffers**, **icetStrategy**

## NAME

**icetCopyState** – copy state machine of one context to another.

## SYNOPSIS

```
#include <GL/ice-t.h>

    void icetCopyState( IceTContext  dest,
                        IceTContext  src  );
```

## DESCRIPTION

The **icetCopyState** function replaces the state of *dest* with the current state of *src*. This function can be used to quickly duplicate a context.

The **IceTCommunicator** object associated with *dest* is *not* changed (nor can it ever be). Consequently, the following state values are not copied either, since they refer to process ids that are directly tied to the **IceTCommunicator** object: **ICET_RANK**, **ICET_NUM_PROCESSES**, **ICET_DATA_REPLICATION_GROUP**, **ICET_DATA_REPLICATION_GROUP_SIZE**, **ICET_COMPOSITE_ORDER**, and **ICET_PROCESS_ORDERS**. However, every other state parameter is copied.

## ERRORS

None.

## WARNINGS

None.

## BUGS

The state is copied blindly. It is therefore possible to copy states that are invalid for a context's communicator. For example, a display rank may not refer to a valid process id.

## NOTES

Behavior is undefined if *dest* or *src* has never been created or has already been destroyed.

## COPYRIGHT



## SEE ALSO

**icetCreateContext**, **icetGetContext**, **icetSetContext**

## NAME

**icetCreateContext** – creates a new context.

## SYNOPSIS

```
#include <GL/ice-t.h>

    IceTContext icetCreateContext( IceTCommunicator comm );
```

## DESCRIPTION

The **icetCreateContext** function creates a new IceT context, makes it current, and returns a handle to the new context. The handle returned is of type **IceTContext**. This is an opaque type that should not be handled directly, but rather simply passed to other IceT functions.

Like OpenGL, the IceT engine behaves like a large state machine. The parameters for engine operation is held in the current state. The entire state is encapsulated in a context. Each new context contains its own state.

It is therefore possible to change the entire current state of IceT by simply switch contexts. Switching contexts is much faster, and often more convenient, than trying to change many state parameters.

## ERRORS

None.

## WARNINGS

None.

## BUGS

It may be tempting to use contexts to run different IceT operations on separate program threads. Although certainly possible, great care must be taken. First of all, all threads will share the same context. Second of all, IceT is not thread safe. Therefore, a multi-threaded program would have to run all IceT commands in 'critical sections' to ensure that the correct context is being used, and the methods execute safely in general.

## NOTES

**icetCreateContext** duplicates the communicator *comm*. Thus, to avoid deadlocks on certain implementations (such as MPI), the user level program should call **icetCreateContext** on all processes with the same *comm* object at about the same time.

## COPYRIGHT



## SEE ALSO

**icetDestroyContext**, **icetGetContext**, **icetSetContext**, **icetCopyState**, **icetGet**

## NAME

**icetCreateMPICommunicator** – Converts an MPI communicator to an IceT communicator.

## SYNOPSIS

```
#include <GL/ice-t_mpi.h>

    IceTCommunicator icetCreateMPICommunicator(
                                            MPI_Comm  mpi_comm  );
```

## DESCRIPTION

IceT requires a communicator in order to perform correctly. An application is free to build its own communicator, but many will simply prefer to use MPI, which is a well established parallel communication tool. Thus, IceT comes with an implementation of **IceTCommunicator** that uses the MPI communication layer underneath.

**icetCreateMPICommunicator** is used to create an **IceTCommunicator** that uses the *mpi_comm* MPI communication object. The resulting **IceTCommunicator** shares the same process group and process rank as the original **MPI_Comm** communicator.

*mpi_comm* is duplicated, which has two consiquences. First, all process in *mpi_comm*'s group may need to call **icetCreateMPICommunicator** in order for any of them to proceed (depending on the MPI implementation). Second, *mpi_comm* and the resulting **IceTCommunicator** are decoupled from each other. Communications in one cannot affect another. Also, one communicator may be destroyed without affecting the other.

## RETURN VALUE

An **IceTCommunicator** with the same process group and rank as *mpi_comm*. The communicator may be destroyed with a call to **icetDestroyMPICommunicator**.

## ERRORS

None.

## WARNINGS

None.

## BUGS

All MPI errors are ignored.

## COPYRIGHT



## SEE ALSO

**`icetDestroyMPICommunicator`**, **`icetCreateContext`**

## NAME

**icetDataReplicationGroup** – define data replication.

## SYNOPSIS

```
#include <GL/ice-t.h>

    void icetDataReplicationGroup( GLint          size,
                                   const GLint *  processes  );
```

## DESCRIPTION

IceT has the ability to take advantage of geometric data that is replicated among processes. If a group of processes share the same geometry data, then IceT will split the region of the display that the data projects onto among the processes, thereby reducing the total amount of image composition work that needs to be done.

Each group can be declared by calling **icetDataReplicationGroup** and defining the group of processes that share the geometry with the local process. *size* indicates how many processes belong to the group, and *processes* is an array of ids of processes that belong to the group. Each process that belongs to a particular group must call **icetDataReplicationGroup** with the exact same list of processes in the same order.

You can alternately use **icetDataReplicationGroupColor** to select data replication groups.

By default, each process belongs to a group of size one containing just the local processes (i.e. there is no data replication).

## ERRORS

**ICET_INVALID_VALUE** *processes* does not contain the local process rank.

## WARNINGS

None.

## BUGS

IceT assumes that **icetDataReplicationGroup** is called with the exact same parameters on all processes belonging to a given group. Likewise, IceT also assumes that all processes have called **icetBoundingVertices** or **icetBoundingBox** with the exact same parameters on all processes belonging to a given group. These requirements are not strictly enforced, but failing to do so may cause some of the geometry to not be rendered.

## COPYRIGHT



## SEE ALSO

**icetDataReplicationGroupColor**, **icetDrawFunc**, **icetBoundingVertices**, **icetBoundingBox**

## NAME

**icetDataReplicationGroupColor** – define data replication.

## SYNOPSIS

```
#include <GL/ice-t.h>

    void icetDataReplicationGroupColor( GLint color );
```

## DESCRIPTION

IceT has the ability to take advantage of geometric data that is replicated among processes. If a group of processes share the same geometry data, then IceT will split the region of the display that the data projects onto among the processes, thereby reducing the total amount of image composition work that needs to be done.

Despite the name of the function, **icetDataReplicationGroupColor** has nothing to do the color of the data being replicated. Instead, *color* is used to mark the local process as part of a given group. When **icetDataReplicationGroupColor** is called, it finds all other processes that have the same color and builds a group based on this information.

**icetDataReplicationGroupColor** must be called on every processes before the function will return.

## ERRORS

None.

## WARNINGS

None.

## BUGS

IceT assumes that **icetDataReplicationGroup** is called with the exact same parameters on all processes belonging to a given group. Likewise, IceT also assumes that all processes have called **icetBoundingVertices** or **icetBoundingBox** with the exact same parameters on all processes belonging to a given group. These requirements are not strictly enforced, but failing

to do so may cause some of the geometry to not be rendered.

## NOTES

This man page should never be installed. It should just be used to help make other man pages.

## COPYRIGHT



## SEE ALSO

**`icetDataReplicationGroup`**, **`icetDrawFunc`**, **`icetBoundingVertices`**, **`icetBoundingBox`**

## NAME

**icetDestroyContext** – delete a context.

## SYNOPSIS

```
#include <GL/ice-t.h>

    void icetDestroyContext( IceTContext context ;
```

## DESCRIPTION

Frees the memory required to hold the state of *context* and removes *context* from existence.

## ERRORS

None.

## WARNINGS

None.

## BUGS

**icetDestroyContext** will happily delete the current context for you, but subsequent calls to most other IceT functions will probably result in seg-faults unless you make another context current with **icetCreateContext** or **icetSetContext**. The most notable execptions are the functions with names matching **icet*Context**, which will work correctly without a proper current context.

## NOTES

Behavior is undefined if *context* has never been created or has already been destroyed.

## COPYRIGHT



## SEE ALSO

**`icetCreateContext`**

## NAME

**icetDestroyMPICommunicator** – deletes a MPI communicator

## SYNOPSIS

```
#include <GL/ice-t_mpi.h>

    void icetDestroyMPICommunicator( IceTCommunicator comm );
```

## DESCRIPTION

Destroys an **IceTCommunicator**. *comm* becomes invalid and any memory or MPI resources held by *comm* are freed.

Communicators are copied when attached to an IceT context, so destroying an **IceTCommunicator** used to create a context still in use is safe.

## ERRORS

None.

## WARNINGS

None.

## BUGS

All MPI errors are ignored.

## COPYRIGHT

## SEE ALSO

**icetCreateMPICommunicator**

## NAME

**icetDiagnostics** – change diagnostic reporting level.

## SYNOPSIS

```
#include <GL/ice-t.h>

    void icetDiagnostics( GLbitfield mask );
```

## DESCRIPTION

Sets what diagnostic message are printed to standard output. The messages to be printed out are defined by *mask*. *mask* consists of flags that are OR-ed together. The valid flags are:

**ICET_DIAG_OFF** A zero flag used to indicate that no diagnostic messages are desired.

**ICET_DIAG_ERRORS** Print messages associated with anomalous conditions.

**ICET_DIAG_WARNINGS** Print messages associated with conditions that are unexpected or may lead to errors. Implicitly turns on **ICET_DIAG_ERRORS**.

**ICET_DIAG_DEBUG** Print frequent messages concerning the status of IceT. Implicitly turns on **ICET_DIAG_ERRORS** and **ICET_DIAG_WARNINGS**.

**ICET_DIAG_ROOT_NODE** Print messages only on the node with a process rank of 0. This is the default if neither **ICET_DIAG_ROOT_NODE** nor **ICET_DIAG_ALL_NODES** is set.

**ICET_DIAG_ALL_NODES** Print messages all every nodes.

**ICET_DIAG_FULL** Turn on all diagnostic messages on all nodes.

The default flags are **ICET_DIAG_ALL_NODES** | **ICET_DIAG_WARNINGS**.

## ERRORS

None.

## WARNINGS

None.

## BUGS

None known.

## COPYRIGHT



## SEE ALSO

**`icetGetError`**

## NAME

**icetDrawFrame** – renders and composites a frame

## SYNOPSIS

```
#include <GL/ice-t.h>

    void icetDrawFrame(void);
```

## DESCRIPTION

Initiates a frame draw using the current OpenGL transformation matrices (modelview and projection).

IceT may render an image, the tile display needs to be defined (using **icetAddTile**) and the drawing function needs to be set (using **icetDrawFunc**). The composite strategy may also optionally be set (using **icetStrategy**).

If **ICET_DISPLAY** is enabled, then the fully composited image is written back to the OpenGL framebuffer for display. It is the application's responsibility to synchronize the processes and swap front and back buffers. If the OpenGL background color is set to something other than black, **ICET_DISPLAY_COLORED_BACKGROUND** should also be enabled. Displaying with **ICET_DISPLAY_COLORED_BACKGROUND** disabled may be slightly faster (depending on graphics hardware) but can result in black rectangles in the background. If **ICET_DISPLAY_INFLATE** is enabled and the size of the renderable window (determined by the current OpenGL viewport) is greater than that of the tile being displayed, then the image will first be 'inflated' to the size of the actual display. If **ICET_DISPLAY_INFLATE** is disabled, the image is drawn at its original resolution at the lower left corner of the display.

The image remaining in the frame buffer is undefined if **ICET_DISPLAY** is disabled or the process is not displaying a tile.

## ERRORS

| | |
|---|---|
| **ICET_INVALID_OPERATION** | Raised if the drawing function has not been set. Also can be raised if **icetDrawFrame** is called recursively, probably from within the drawing callback. |
| **ICET_OUT_OF_MEMORY** | Not enough memory left to hold intermittent frame buffers and other temporary data. |

## WARNINGS

None.

## BUGS

If compositing with color blending on, the image returned with **icetGetColorBuffer** may have values of $\langle R,G,B,A\rangle = \langle 0,0,0,0\rangle$ and the rest of the image may be blended with black rather than the correct background color.

During compositing, image compression is employed that relies on knowing the maximum possible value in the z-buffer. Unfortunately, different rendering hardware can give different results for this value. IceT tries to dermine this value up front by clearing the screen and reading the z-buffer value, but this test sometimes fails, resulting in a classification of background. The side effects of this are minimal, and IceT usually quickly fixes the problem by continually checking depth values.

## COPYRIGHT



## SEE ALSO

**icetResetTiles**, **icetAddTile**, **icetBoundingBox**, **icetBoundingVertices**, **icetDrawFunc**, **icetStrategy**, **icetGetColorBuffer**, **icetGetDepthBuffer**

## NAME

**icetDrawFunc** – set a callback for drawing.

## SYNOPSIS

```
#include <GL/ice-t.h>

   typedef void (*IceTCallback)(void);

    void icetDrawFunc(  IceTCallback  func  );
```

## DESCRIPTION

The **icetDrawFunc** function sets a callback that is used to draw the geometry from a given viewpoint.

*func* should be a function that issues appropriate OpenGL calls to draw geometry in the current OpenGL context. After *func* is called, the image left in the back frame buffer will be read back for compositing.

*func* should *not* modify the **GL_PROJECTION_MATRIX** as this would cause IceT to place image data in the wrong location in the tiled display and improperly cull geometry. It is acceptable to add transformations to **GL_MODELVIEW_MATRIX**, but the bounding vertices given with **icetBoundingVertices** or **icetBoundingBox** are assumed to already be transformed by any such changes to the modelview matrix. Also, **GL_MODELVIEW_MATRIX** must be restored before the draw function returns. Therefore, any changes to **GL_MODELVIEW_MATRIX** are to be done with care and should be surrounded by a pair of glPushMatrix and glPopMatrix functions.

It is also important that *func* *not* attempt the change the clear color. In some composting modes, IceT needs to read, modify, and change the background color. These operations will be lost if *func* changes the background color, and severe color blending artifacts may result.

IceT may call *func* several times from within a call to **icetDrawFrame** or not at all if the current bounds lie outside the current viewpoint. This can have a subtle but important impact on the behavior of *func*. For example, counting frames by incrementing a frame counter in *func* is obviously wrong (although you could count how many times a render occurs). *func* should also leave OpenGL in a state such that it will be correct for a subsequent run of *func*. Any matrices or attributes pushed in *func* should be popped before *func* returns, and any state that is assumed to be true on entrance to *func* should also be true on return.

The *func* function pointer is placed in the **ICET_DRAW_FUNCTION** state variable.

## ERRORS

None.

## WARNINGS

None.

## BUGS

None known.

## NOTES

*func* is tightly coupled with the bounds set with **icetBoundingVertices** or **icetBoundingBox**. If the geometry drawn by *func* is dynamic (changes from frame to frame), then the bounds may need to be changed as well. Incorrect bounds may cause the geometry to be culled in surprising ways.

## COPYRIGHT



## SEE ALSO

**icetDrawFrame**, **icetBoundingVertices**, **icetBoundingBox**

## NAME

**icetEnable**, **icetDisable**– enable/disable an IceT feature.

## SYNOPSIS

```
#include <GL/ice-t.h>

    void icetEnable   ( GLenum pname );
    void icetDisable  ( GLenum pname );
```

## DESCRIPTION

The **icetEnable** and **icetDisable** functions turn various IceT features on and off. *pname* is a symbolic constant representing the feature to be turned on or off. Valid values for *pname* are:

**ICET_CORRECT_COLORED_BACKGROUND** Colored backgrounds are problematic when performing color blended compositing in that the background color will be additively blended from each image. Enabling this flag will internally cause the color to be reset to black and then cause the color to be blended back into the resulting images. This flag is disabled by default.

**ICET_DISPLAY** If enabled, the final, composited image for each tile is written back to the frame buffer before the return of **icetDrawFrame**. This flag is enabled by default.

**ICET_DISPLAY_COLORED_BACKGROUND** If this and **ICET_DISPLAY** are enabled, uses OpenGL blending to ensure that all background is set to the correct color. This flag is disabled by default. This option does not affect the images returned from **icetGetColorBuffer** or **icetGetDepthBuffer**; it only affects the image in the OpenGL color buffer.

**ICET_DISPLAY_INFLATE** If this and **ICET_DISPLAY** are enabled and the renderable window is larger then the displayed tile (as determined by the current OpenGL viewport), then resample the image to fit within the renderable window before writing back to frame buffer. This flag is disabled by default. This option does not affect the images returned from **icetGetColorBuffer** or **icetGetDepthBuffer**; it only affects the image in the OpenGL color buffer.

**ICET_DISPLAY_INFLATE_WITH_HARDWARE** This option determines how images are inflated. When enabled (the default), images are inflated by creating a texture and allowing the hardware the inflate the image. When disabled, images are inflated on the CPU. This option has no effect unless both **ICET_DISPLAY** and **ICET_DISPLAY_INFLATE** are also enabled.

**ICET_FLOATING_VIEWPORT** If enabled, the projection will be shifted such that the geometry will be rendered in one shot whenever possible, even if the geometry straddles up to four tiles. This flag is enabled by default.

**ICET_ORDERED_COMPOSITE** If enabled, the image composition will be performed in the order specified by the last call to **icetCompositeOrder**, assuming the current strategy (specified by a call to **icetStrategy**) supports ordered composition. Generally, you want to enable ordered compositing if doing color blending and disable if you are doing z-buffer comparisons. If enabled, you should call **icetCompositeOrder** between each frame to update the image order as camera angles change. This flag is disabled by default.

## ERRORS

**ICET_INVALID_VALUE** If *pname* is not a feature to be enabled or disabled.

## WARNINGS

None.

## BUGS

The check for a valid *pname* is not thorough, and thus the **ICET_INVALID_VALUE** error may not always be raised.

## COPYRIGHT



## SEE ALSO

**icetIsEnabled**

## NAME

**icetGet** – get an IceT state parameter

## SYNOPSIS

```
#include <GL/ice-t.h>

    void icetGetDoublev ( GLenum     pname,
                          GLdouble * params  );

    void icetGetFloatv ( GLenum     pname,
                         GLfloat *  params  );

    void icetGetIntegerv ( GLenum   pname,
                           GLint *  params  );

    void icetGetBooleanv ( GLenum      pname,
                           GLboolean * params  );

    void icetGetPointerv ( GLenum     pname,
                           GLvoid **  params  );
```

## DESCRIPTION

Like OpenGL, the operation of IceT is defined by a large state machine. Also like OpenGL, the state parameters can be retrieved through the **icetGet** functions. Each function takes a symbolic constant, *pname*, which identifies the state parameter to retrieve. They also each take an array, *params*, which will be filled with the values in *pname*. It is the calling application's responsibility to ensure that *params* is big enough to hold all the data.

## STATE PARAMETERS

The following list identifies valid values for *pname* and a description of the associated state parameter.

**ICET_ABSOLUTE_FAR_DEPTH** The maximum possible value in the depth buffer (i.e. the value in a cleared depth buffer), as stored as an unsigned 32 bit integer. Usually, this is the expected `0xFFFFFFFF`. However, some systems that use buffer values with 24 bits or less cast the maximum value to something smaller.

**ICET_BACKGROUND_COLOR** The color that IceT is currently assuming is the background color.

It is an RGBA value that is stored as four floating point values. This value is generally taken from the OpenGL background color on a call to **icetDrawFrame**, but is also occasionally set to black to make sure that color blending happens correctly. (The correct background color is restored later.)

**ICET_BACKGROUND_COLOR_WORD** The same as **ICET_BACKGROUND_COLOR** except that each component is stored as 8-bit values and packed in a 4-byte integer as specified by **ICET_COLOR_FORMAT**. The idea is to rapidly fill the background of color buffers. This value is generally taken from the OpenGL background color on a call to **icetDrawFrame**, but is also occasionally set to black to make sure that color blending happens correctly. (The correct background color is restored later.)

**ICET_BLEND_TIME** The total time, in seconds, spent in performing color blending of images during the last call to **icetDrawFrame**. Stored as a double. An alias for this value is **ICET_COMPARE_TIME**.

**ICET_BUFFER_READ_TIME** The total time, in seconds, spent reading from OpenGL buffers during the last call to **icetDrawFrame**. Stored as a double.

**ICET_BUFFER_WRITE_TIME** The total time, in seconds, spent writing to OpenGL buffers during the last call to **icetDrawFrame**. Stored as a double.

**ICET_BYTES_SENT** The total number of bytes sent by the calling process for transferring image data during the last call to **icetDrawFrame**. Stored as an integer.

**ICET_COLOR_BUFFER_VALID** True if a color buffer was computed during the last call to **icetDrawFrame** and is available with a call to **icetGetColorBuffer**.

**ICET_COLOR_FORMAT** The OpenGL symbolic constant describing the format in which IceT reads and stores color buffers. Currently always set to **GL_RGBA**, **GL_BGRA**, or **GL_BGRA_EXT**.

**ICET_COMPARE_TIME** The total time, in seconds, spent in performing Z comparisons of images during the last call to **icetDrawFrame**. Stored as a double. An alias for this value is **ICET_BLEND_TIME**.

**ICET_COMPOSITE_ORDER** The order in which images are to be composited if **ICET_ORDERED_COMPOSITE** is enabled and the current startegy supports ordered compositing. The parameter contains **ICET_NUM_PROCESSES** entries. The value of this parameter is set with **icetCompositeOrder**. If the element of index $i$ in the array is set to $j$, then there are $i$ images "on top" of the image generated by process $j$.

**ICET_COMPOSITE_TIME** The total time, in seconds, spent in compositing during the last call to **icetDrawFrame**. Equal to **ICET_TOTAL_DRAW_TIME** − **ICET_RENDER_TIME** − **ICET_BUFFER_READ_TIME** − **ICET_BUFFER_WRITE_TIME**. Stored as a double.

**ICET_COMPRESS_TIME** The total time, in seconds, spent in compressing image data using active pixel encoding during the last call to **icetDrawFrame**. Stored as a double.

**ICET_DATA_REPLICATION_GROUP** An array of process ids. There are **ICET_DATA_REPLICATION_GROUP_SIZE** entries in the array. IceT assumes that all processes in the list will create the exact same image with their draw functions (set with **icetDrawFunc**). The local process id (**ICET_RANK**) will be part of this list.

**ICET_DATA_REPLICATION_GROUP_SIZE** The length of the **ICET_DATA_REPLICATION_GROUP** array.

**ICET_DEPTH_BUFFER_VALID** True if a depth buffer was computed during the last call to **icetDrawFrame** and is available with a call to **icetGetDepthBuffer**.

**ICET_DIAGNOSTIC_LEVEL** The diagnostics flags set with **icetDiagnostics**.

**ICET_DISPLAY_NODES** An array of process ranks. The size of the array is equal to the number of tiles (**ICET_NUM_TILES**). The $i^{\text{th}}$ entry is the rank of the process that is displaying the tile described by the $i^{\text{th}}$ entry in **ICET_TILE_VIEWPORTS**.

**ICET_DRAW_FUNCTION** A pointer to the drawing callback function, as set by **icetDrawFunc**.

**ICET_INPUT_BUFFERS** A bitmask specifying the the buffers which IceT will read from OpenGL and perform composition. The value is set with **icetInputOutputBuffers**. See the documentation of that function for valid bit flags.

**ICET_FRAME_COUNT** The number of times **icetDrawFrame** has been called for the current context.

**ICET_GEOMETRY_BOUNDS** An array of vertices whose convex hull bounds the drawn geometry. Set with **icetBoundingVertices** or **icetBoundingBox**. Each vertex has three coordinates and are tightly packed in the array. The size of the array is $3 \times$ **ICET_NUM_BOUNDING_VERTS**.

**ICET_GLOBAL_VIEWPORT** Defines a viewport in an infinite logical display that covers all tile viewports (listed in **ICET_TILE_VIEWPORTS**). The viewport, like an OpenGL viewport, is given as the integer four-tuple $\langle x, y, width, height \rangle$. $x$ and $y$ are placed at the leftmost and lowest position of all the tiles, and *width* and *height* are just big enough for the viewport to cover all tiles. The viewports are listed in the same order as the tiles were defined with **icetAddTile**.

**ICET_NUM_BOUNDING_VERTS** The number of bounding vertices listed in the **ICET_GEOMETRY_BOUNDS** parameter.

**ICET_NUM_TILES** The number of tiles in the defined display. Basically equal to the number of times **icetAddTile** was called after the last **icetResetTiles**.

**ICET_NUM_PROCESSES** The number of processes in the parallel job as given by the **IceTCommunicator** object associated with the current context.

**ICET_OUTPUT_BUFFERS** A bitmask specifying the the buffers which IceT will generate from composition. The value is set with **icetInputOutputBuffers**. See the documentation of that function for valid bit flags.

**ICET_PROCESS_ORDERS** Basically, the inverse of **ICET_COMPOSITE_ORDER**. The parameter contains **ICET_NUM_PROCESSES** entries. If the element of index $i$ in the array is set to $j$, then there are $j$ images "on top" of the image generated by process $i$.

**ICET_RANK** The rank of the process as given by the **IceTCommunicator** object associated with the current context.

**ICET_READ_BUFFER** Set to the OpenGL symbolic constant that IceT will use to read back buffers. Currently always set to **GL_BACK**.

**ICET_RENDER_TIME** The total time, in seconds, spent in the drawing callback during the last call to **icetDrawFrame**. Stored as a double.

**ICET_STRATEGY_SUPPORTS_ORDERING** Is true if and only if the current strategy supports ordered compositing.

**ICET_TILE_DISPLAYED** The index of the tile the local process is displaying. The index will correspond to the tile entry in the **ICET_DISPLAY_NODES** and **ICET_TILE_VIEWPORTS** arrays. If set to $0 <= i <$ **ICET_NUM_PROCESSES**, then the $i^{\text{th}}$ entry of **ICET_DISPLAY_NODES** is equal to **ICET_RANK**. If the local process is not displaying any tile, then **ICET_TILE_DISPLAYED** is set to $-1$.

**ICET_TILE_MAX_HEIGHT** The maximum *height* of any tile.

**ICET_TILE_MAX_PIXELS** The maximum number of pixels in any tile. This number is actually set to (**ICET_TILE_MAX_WIDTH** $\times$ **ICET_TILE_MAX_HEIGHT**) + **ICET_NUM_PROCESSES**. The number of processes is added to provide sufficient padding such that the max tile image may be divided evenly amongst any group of processes without dropping any real pixels.

**ICET_TILE_MAX_WIDTH** The maximum *width* of any tile.

**ICET_TILE_VIEWPORTS** A list of viewports in the logical global display defining the tiles. Each viewport is the four-tuple $\langle x, y, width, height \rangle$ defining the position and dimensions of a tile in pixels, much like a viewport is defined in OpenGL. The size of the array is $4 *$ **ICET_NUM_TILES**.

**ICET_TOTAL_DRAW_TIME** Time spent in the last call to **icetDrawFrame**. Stored as a double.

## ERRORS

| | |
|---|---|
| **ICET_BAD_CAST** | The state parameter requested is of a type that cannot be cast to the output type. |
| **ICET_INVALID_ENUM** | *pname* is not a valid state parameter. |

## WARNINGS

None.

## BUGS

None known.

## NOTES

Not every state variable is documented here. There is a set of parameters used internally by IceT or are more appropriately retrieved with other functions such as **`icetIsEnabled`**.

## COPYRIGHT



## SEE ALSO

**`icetIsEnabled`**, **`icetGetStrategyName`**

## NAME

**icetGetColorBuffer**, **icetGetDepthBuffer**– retrieves the last computed color or depth buffer.

## SYNOPSIS

```
#include <GL/ice-t.h>

    GLubyte  *icetGetColorBuffer  ( void );
    GLuint   *icetGetDepthBuffer  ( void );
```

## DESCRIPTION

Returns a buffer containing the result of the image composition performed by the last call to **icetDrawFrame**. Be aware that a color or depth buffer may not have been computed with the last call to **icetDrawFrame**. IceT avoids the computation and network transfers for any unnecessary buffers unless specifically requested otherwise with the flags given to the **icetInputOutputBuffers** function. Use a call to **icetGetBooleanv** with a value of **ICET_COLOR_BUFFER_VALID** or **ICET_DEPTH_BUFFER_VALID** to determine whether either of these buffers are available. Attempting to get a nonexistent buffer will result with a warning being emitted and `NULL` returned.

## RETURN VALUE

**icetGetColorBuffer** returns the color buffer for the displayed tile. Each pixel value can be assumed to be four consecutive bytes in the buffer. The pixels are also always aligned on 4-byte boundaries. The format of the color buffer is defined by the state parameter **ICET_COLOR_FORMAT**, which is typically either **GL_RGBA**, **GL_BGRA**, or **GL_BGRA_EXT**.

**icetGetDepthBuffer** returns the depth buffer for the displayed tile. Depth values are stored as 32-bit integers.

The width and the height of the buffer are determined by the width and the height of the displayed tile at the time **icetDrawFrame** was called. If the tile layout is changed since the last call to **icetDrawFrame**, the dimensions of the buffer returned may not agree with the dimensions stored in the current IceT state.

The memory returned by **icetGetColorBuffer** and **icetGetDepthBuffer** need not, and should not, be freed. It will be reclaimed in the next call to **icetDrawFrame**. Expect the data returned to be obliterated on the next call to **icetDrawFrame**.

## ERRORS

None.

## WARNINGS

| | |
|---|---|
| **ICET_INVALID_VALUE** | The appropriate buffer is not available, either because it was not computed or it has been obliterated by a subsequent IceT computation. |

## BUGS

The returned image may have a value of $(R,G,B,A) = (0,0,0,0)$ for a pixel instead of the true background color. This can usually be corrected by replacing all pixels with an alpha value of 0 with the background color.

The buffers are stored in a shared memory pool attached to a particular context. As such, the buffers are not copied with the state. Also, because they are shared, it is conceivable that the buffers will be reclaimed before the next call to **icetDrawFrame**. If this should happen, the **ICET_COLOR_BUFFER_VALID** and **ICET_DEPTH_BUFFER_VALID** state variables will be set accordingly.

## COPYRIGHT



## SEE ALSO

**icetDrawFrame**, **icetInputOutputBuffers**, **icetGet**

## NAME

**icetGetContext** – retrieves the current context

## SYNOPSIS

```
#include <GL/ice-t.h>

    IceTContext icetGetContext( void );
```

## DESCRIPTION

The **icetGetContext** function retrieves the handle for the current context. This handle may be stored and set for later use with **icetSetContext** (assuming the context has not been since destroyed).

## RETURN VALUE

A handle for the current context.

## ERRORS

None.

## WARNINGS

None.

## BUGS

None known.

## COPYRIGHT

## SEE ALSO

**icetSetContext**, **icetCreateContext**, **icetDestroyContext**, **icetCopyState**

## NAME

**icetGetError** – return the last error condition.

## SYNOPSIS

```
#include <GL/ice-t.h>

    GLenum icetGetError( void );
```

## DESCRIPTION

Retrieves the first error or warning condition that occurred since the last call to **icetGetError** or since program startup, whichever happened last.

Once an error condition has been retrieved with **icetGetError**, the error condition is reset to no error and cannot be retrieved again.

## RETURN VALUE

One of the following flags will be returned:

| | |
|---|---|
| **ICET_INVALID_VALUE** | An inappropriate value has been passed to a function. |
| **ICET_INVALID_OPERATION** | An inappropriate function has been called. |
| **ICET_OUT_OF_MEMORY** | IceT has ran out of memory for buffer space. |
| **ICET_BAD_CAST** | A function has been passed a value of the wrong type. |
| **ICET_INVALID_ENUM** | A function has been passed an invalid constant. |
| **ICET_SANITY_CHECK_FAIL** | An internal error (or warning) has occurred. |
| **ICET_NO_ERROR** | No error has been raised since the last call to **icetGetError**. |

## BUGS

It is not possible to tell if the returned value was caused by an error or a warning.

## NOTES

The error value is *not* context dependent.

## COPYRIGHT



## SEE ALSO

**`icetDiagnostics`**

## NAME

**icetGetStrategyName** – retrieve strategy name.

## SYNOPSIS

```
#include <GL/ice-t.h>

    const GLubyte icetGetStrategyName(  void  );
```

## DESCRIPTION

**icetGetStrategyName** retrieves a human readable name for the current strategy.

## RETURN VALUE

Returns a short, null terminated string identifying the strategy currently in effect. Helpful for printing out debugging or diagnostic statements. If no strategy is set, `NULL` is returned.

## ERRORS

None.

## WARNINGS

None.

## BUGS

None known.

## NOTES

The string returned does *not* contain the identifier used in a C program. For example, if the current strategy is **ICET_STRATEGY_REDUCE**, **icetGetStrategyName** returns "Reduce," not "ICET_STRATEGY_REDUCE."

## COPYRIGHT



## SEE ALSO

**`icetStrategy`**

## NAME

**icetInputOutputBuffers** – set IceT composition mode.

## SYNOPSIS

```
#include <GL/ice-t.h>

    void icetInputOutputBuffers( GLenum inputs,
                                 GLenum outputs );
```

## DESCRIPTION

**icetInputOutputBuffers** sets what OpenGL frame buffers IceT reads and generates. During a call to **icetDrawFrame**, IceT reads the input buffers directly from OpenGL after it performs a callback to the draw function (set by **icetDrawFunc**). Output buffers are stored internally after the call to **icetDrawFrame** finishes. The output buffers can be retrieved with calls to the **icetGetColorBuffer** and **icetGetDepthBuffer** functions. In addition, if the color buffer output is on and **ICET_DISPLAY** is enabled, the color buffer is also written back to the OpenGL frame buffer before **icetDrawFrame** returns.

Both *inputs* and *outputs* are or'ed values of one or more of the following flags:

**ICET_COLOR_BUFFER_BIT** Reads/generates color data. Color data is stored in RGBA or BGRA format. Each channel is 8-bits, resulting in a 32-bit word when combined together. Each 32-bit color value is always aligned on 32-bit word boundaries for faster computation.

**ICET_DEPTH_BUFFER_BIT** Reads/generates depth data. Depth data is stored as 32-bit unsigned integers.

The current values of the input and output buffers are stored in the **ICET_INPUT_BUFFERS** and **ICET_OUTPUT_BUFFERS** state variables. By default, the **ICET_INPUT_BUFFERS** value is set to (**ICET_COLOR_BUFFER_BIT**|**ICET_DEPTH_BUFFER_BIT**), and the **ICET_OUTPUT_BUFFERS** value is set to **ICET_COLOR_BUFFER_BIT**.

The composition operator IceT uses is defined by the inputs. If the depth buffer is an input, then Z comparison is performed. If the depth buffer is not an input, alpha blending is performed. Note that in the latter case, order of composition may matter and therefore not all composition strategies will work.

## ERRORS

| | |
|---|---|
| **ICET_INVALID_VALUE** | An output was selected that is not also an input or no outputs were selected at all. |

## WARNINGS

None.

## BUGS

Blending of colors cannot be used in conjunction with depth testing. Even with depth testing, the order of operation for color blending is important, so such a combination is not likely to be useful.

## COPYRIGHT



## SEE ALSO

**icetGetColorBuffer**, **icetGetDepthBuffer**, **icetDrawFrame**

## NAME

**icetIsEnabled** – query enabled status of an IceT feature.

## SYNOPSIS

```
#include <GL/ice-t.h>

    GLboolean icetIsEnabled( GLenum pname );
```

## RETURN VALUE

Returns **ICET_TRUE** if the feature associated with *pname* is enabled, **ICET_FALSE** ($= 0$) if the feature is disabled.

## ERRORS

**ICET_INVALID_VALUE** If *pname* is not a feature to be enabled or disabled.

## WARNINGS

None.

## BUGS

The check for a valid *pname* is not thorough, and thus the **ICET_INVALID_VALUE** error may not always be raised.

## NOTES

A list of valid values for *pname* is given in the documentation for **icetEnable**.

## COPYRIGHT

## SEE ALSO

**icetEnable**

## NAME

**icetResetTiles** – clears out all tile definitions.

## SYNOPSIS

```
#include <GL/ice-t.h>

    void icetResetTiles( void )
```

## DESCRIPTION

IceT defines its display as a set of tiles. **icetResetTiles** will empty this set. The set of tiles is filled again with calls to **icetAddTile**.

As a side effect, **icetResetTiles** will also zero out the renderable window size. The size will be reset with calls to **icetAddTile**.

## ERRORS

None.

## WARNINGS

None.

## BUGS

None known.

## NOTES

As a rule, a call to **icetResetTiles** should always be followed with one or more calls to **icetAddTile**. **icetDrawFrame** will not work properly if no tiles are in existence.

## COPYRIGHT



## SEE ALSO

**`icetAddTile`**

## NAME

**icetSetContext** – changes the current context.

## SYNOPSIS

```
#include <GL/ice-t.h>

    void icetSetContext( IceTContext context );
```

## DESCRIPTION

The **icetSetContext** function sets the IceT state machine to work with the context defined by *context* and the state associated with it. Further calls to IceT functions will operate based on the state encapsulated in *context*. Changing the state of the context is a fast operation.

## ERRORS

**ICET_INVALID_VALUE** *context* is not valid.

## WARNINGS

None.

## BUGS

None known.

## NOTES

The behavior of **icetSetContext** is somewhat indeterminate if *context* is not valid. Usually, an **ICET_INVALID_VALUE** error will be raised, but it is possible that the context will be set to some other context. Under any circumstances, a valid context will be current when this function returns.

## COPYRIGHT



## SEE ALSO

**icetGetContext**, **icetCreateContext**, **icetCopyState**

## NAME

**icetStrategy** – set the strategy used to composite images.

## SYNOPSIS

```
#include <GL/ice-t.h>

    void icetStrategy( IceTStrategy strategy );
```

## DESCRIPTION

The IceT API comes packaged with several algorithms for compositing images. The algorithm to use is determined by selecting a *strategy*. The strategy is selected with **icetStrategy**. A strategy must be selected before **icetDrawFrame** is called.

The *strategy* is of type **IceTStrategy**. This is an opaque type. There are no conventions to create or change an **IceTStrategy**, but there are several predefined strategies to select from. They are:

**ICET_STRATEGY_SERIAL** Basically applies a "traditional" single tile composition (such as binary swap) to each tile in the order they were defined. Because each process must take part in the composition of each tile regardless of whether they draw into it, this strategy is usually very inefficient when compositing for more than tile. It is provided mostly for comparative purposes.

**ICET_STRATEGY_DIRECT** As each process renders an image for a tile, that image is sent directly to the process that will display that tile. This usually results in a few processes receiving and processing the majority of the data, and is therefore usually an inefficient strategy.

**ICET_STRATEGY_SPLIT** Like **ICET_STRATEGY_DIRECT**, except that the tiles are split up, and each process is assigned a piece of a tile in such a way that each process receives and handles about the same amount of data. This strategy is often very efficient, but due to the large amount of messages passed, it has not proven to be very scalable or robust.

**ICET_STRATEGY_REDUCE** A two phase algorithm. In the first phase, tile images are redistributed such that each process has one image for one tile. In the second phase, a "traditional" single tile composition is performed for each tile. Since each process contains an image for only one tile, all these compositions may happen simultaneously. This is a well rounded strategy that seems to perform well in a wide variety of applications.

**ICET_STRATEGY_VTREE** An extension to the binary tree algorithm for image composition. Sets up a 'virtual' composition tree for each tile image. Processes that belong to multiple trees (because they render to more than one tile) are allowed to float between trees. This strategy is not quite as well load balanced as **ICET_STRATEGY_REDUCE** or **ICET_STRATEGY_SPLIT**, but has very well behaved network communication.

Not all of the strategies support ordered image composition. **ICET_STRATEGY_SERIAL**, **ICET_STRATEGY_DIRECT**, and **ICET_STRATEGY_REDUCE** do support ordered image composition. **ICET_STRATEGY_SPLIT** and **ICET_STRATEGY_VTREE** do not support ordered image composition and will ignore **ICET_ORDERED_COMPOSITE** if it is enabled.

## ERRORS

None.

## WARNINGS

None.

## BUGS

Use the **ICET_STRATEGY_SPLIT** strategy with care. It has proven to be unreliable on several high-speed interconnects. Avoid using it at all in a production application.

## COPYRIGHT



## SEE ALSO

**icetDrawFrame**, **icetGetStrategyName**

## NAME

**icetWallTime** – timer function

## SYNOPSIS

```
#include <GL/ice-t.h>

    double icetWallTime( void )
```

## DESCRIPTION

Retrieves the current time, in seconds. The returned values of **icetWallTime** are only valid in relation to each other. That is, the time may or may not have anything to do with the current date or time. However, the difference of values between two calls to **icetWallTime** is the elapsed time in seconds between the two calls. Thus, **icetWallTime** is handy for determining the running time of various subprocesses. **icetWallTime** is used internally for determining the values for the state variables **ICET_BUFFER_READ_TIME**, **ICET_BUFFER_WRITE_TIME**, **ICET_COMPARE_TIME**, **ICET_COMPOSITE_TIME**, **ICET_COMPRESS_TIME**, **ICET_RENDER_TIME**, and **ICET_TOTAL_DRAW_TIME**.

## RETURN VALUE

The current time, in seconds.

## ERRORS

| | |
|---|---|
| **ICET_INVALID_VALUE** | You've screwed up something a little bit. |
| **ICET_INVALID_OPERATION** | You've screwed something up a lot. |
| **ICET_OUT_OF_MEMORY** | You need a better computer to do what you want to do. |
| **ICET_BAD_CAST** | The argument is of the wrong format. |
| **ICET_INVALID_ENUM** | One of us does not know what he is talking about. |
| **ICET_SANITY_CHECK_FAIL** | I've screwed something up a lot. |

## WARNINGS

None.

## BUGS

None known.

## NOTES

This man page should never be installed. It should just be used to help make other man pages.

## COPYRIGHT



## SEE ALSO

**icetGet**

# Index

# DISTRIBUTION:

3 Berk Geveci
Kitware, Inc.
28 Corporate Drive
Clifton Park, NY 12065

8 MS 1323 Kenneth Moreland, 1424
1 MS 0899 Technical Library, 9536 (electronic)

Sandia National Laboratories